The Patent Office
Concept House
Cardiff Road
Newport
South Wales
NP10 8QQ

I, the undersigned, being an officer duly authorised in accordance with Section 74(1) and (4) of the Deregulation & Contracting Out Act 1994, to sign and issue certificates on behalf of the Comptroller-General, hereby certify that annexed hereto is a true copy of the documents as originally filed in connection with the patent application identified therein.

In accordance with the Patents (Companies Re-registration) Rules 1982, if a company named in this certificate and any accompanying documents has re-registered under the Companies Act 1980 with the same name as that with which it was registered immediately before re-registration save for the substitution as, or inclusion as, the last part of the name of the words "public limited company" or their equivalents in Welsh, references to the name of the company in this certificate and any accompanying documents shall be treated as references to the name with which it is so re-registered.

In accordance with the rules, the words "public limited company" may be replaced by p.l.c., plc, P.L.C. or PLC.

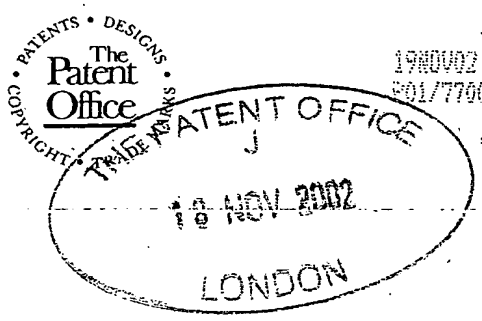
Re-registration under the Companies Act does not constitute a new legal entity but merely subjects the company to certain additional company law rules.

Signed

Dated 28 October 2003

Patents Form 1/77

Pate[illegible]ct 1977
(Rule 16)



19NOV02 E744464-1 002246
P01/7700 0.00-0226908.2

Request for grant of a patent

(See the notes on the back of this form. You can also get an explanatory leaflet from the Patent Office to help you fill in this form)

The Patent Office

Cardiff Road
Newport
South Wales
NP10 8QQ

1. Your reference

P015377GB

2. Patent application number
(The Patent Office will fill in this part)

0226908.2

3. Full name, address and postcode of the or of each applicant *(underline all surnames)*

ARM LIMITED
110 Fulbourn Road
Cherry Hinton
Cambridge
CB1 9NJ

Patents ADP number *(if you know it)*

7498124002

If the applicant is a corporate body, give the country/state of its incorporation

United Kingdom

4. Title of the invention

EXCEPTION HANDLING CONTROL IN A SECURE PROCESSING SYSTEM

5. Name of your agent *(if you have one)*

D Young & Co

"Address for service" in the United Kingdom to which all correspondence should be sent *(including the postcode)*

21 New Fetter Lane
London
EC4A 1DA

Patents ADP number *(if you know it)*

59006

6. If you are declaring priority from one or more earlier patent applications, give the country and the date of filing of the or of each of these earlier applications and *(if you know it)* the or each application number

Country	Priority application number *(if you know it)*	Date of filing *(day / month / year)*

7. If this application is divided or otherwise derived from an earlier UK application, give the number and the filing date of the earlier application

Number of earlier application	Date of filing *(day / month / year)*

8. Is a statement of inventorship and of right to grant of a patent required in support of this request? *(Answer 'Yes' if:*

a) any applicant named in part 3 is not an inventor, or
b) there is an inventor who is not named as an applicant, or
c) any named applicant is a corporate body.
See note (d))

Yes

Patents Form 1/77

9. Enter the number of sheets for any of the following items you are filing with this form. Do not count copies of the same document

Continuation sheets of this form	0
Description	71
Claim *(s)*	4
Abstract	1
Drawing *(s)*	39

10. If you are also filing any of the following, state how many against each item.

Priority documents	0
Translations of priority documents	0
Statement of inventorship and right to grant of a patent *(Patents Form 7/77)*	0
Request for preliminary examination and search *(Patents Form 9/77)*	1
Request for substantive examination *(Patents Form 10/77)*	0
Any other documents *(please specify)*	0

11. I/We request the grant of a patent on the basis of this application.

Signature D Young & Co (Agents for the Applicants) Date 18 Nov 2002

12. Name and daytime telephone number of person to contact in the United Kingdom: Nigel Robinson 023 8071 9500

Warning

After an application for a patent has been filed, the Comptroller of the Patent Office will consider whether publication or communication of the invention should be prohibited or restricted under Section 22 of the Patents Act 1977. You will be informed if it is necessary to prohibit or restrict your invention in this way. Furthermore, if you live in the United Kingdom, Section 23 of the Patents Act 1977 stops you from applying for a patent abroad without first getting written permission from the Patent Office unless an application has been filed at least 6 weeks beforehand in the United Kingdom for a patent for the same invention and either no direction prohibiting publication or communication has been given, or any such direction has been revoked.

Notes

a) If you need help to fill in this form or you have any questions, please contact the Patent Office on 08459 500505.

b) Write your answers in capital letters using black ink or you may type them.

c) If there is not enough space for all the relevant details on any part of this form, please continue on a separate sheet of paper and write "see continuation sheet" in the relevant part(s). Any continuation sheet should be attached to this form.

d) If you have answered 'Yes' Patents Form 7/77 will need to be filed.

e) Once you have filled in the form you must remember to sign and date it.

f) For details of the fee and ways to pay please contact the Patent Office.

DUPLICATE

EXCEPTION HANDLING CONTROL IN A SECURE PROCESSING SYSTEM

This invention relates to data processing systems. More particularly, this invention relates to the control of exception processing within data processing systems.

A data processing apparatus will typically include a processor for running applications loaded onto the data processing apparatus. The processor will operate under the control of an operating system. The data required to run any particular application will typically be stored within a memory of the data processing apparatus. It will be appreciated that the data may consist of the instructions contained within the application and/or the actual data values used during the execution of those instructions on the processor.

There arise many instances where the data used by at least one of the applications is sensitive data that should not be accessible by other applications that can be run on the processor. An example would be where the data processing apparatus is a smart card, and one of the applications is a security application which uses sensitive data, such as for example secure keys, to perform validation, authentication, decryption and the like. It is clearly important in such situations to ensure that such sensitive data is kept secure so that it cannot be accessed by other applications that may be loaded onto the data processing apparatus, for example hacking applications that have been loaded onto the data processing apparatus with the purpose of seeking to access that secure data.

In known systems, it has typically been the job of the operating system developer to ensure that the operating system provides sufficient security to ensure that the secure data of one application cannot be accessed by other applications running under the control of the operating system. However, as systems become more complex, the general trend is for operating systems to become larger and more complex, and in such situations it becomes increasingly difficult to ensure sufficient security within the operating system itself.

Examples of systems seeking to provide secure storage of sensitive data and to provide protection against malicious program code are those described in United States Patent Application US 2002/0007456 A1 and United States Patents US 6,282,657 B and US 6,292,874 B.

Accordingly, it will be desirable to provide an improved technique for seeking to retain the security of such secure data contained within the memory of the data processing apparatus.

Viewed from one aspect the invention provides apparatus for processing data, said apparatus comprising:

a processor operable in a plurality of modes and either a secure domain or a non-secure domain including:

at least one secure mode being a mode in said secure domain; and

at least one non-secure mode being a mode in said non-secure domain; wherein

when said processor is executing a program in a secure mode said program has access to secure data which is not accessible when said processor is operating in a non-secure mode;

said processor is responsive to one or more exception conditions for triggering exception processing; and

said processor being responsive to one or more parameters specifying which of said exceptions should be handled by a secure mode exception handler executing in a secure mode and which of said exceptions should be handled by an exception handler executing in a mode within a current one of said secure domain and said non-secure domain when that exception occurs.

The invention recognises that the way in which a secure processing system having a secure domain and a non-secure domain deals with exceptions is important in preserving the overall security of the system whilst ensuring that the system is able to cope with the requirements of different operating systems. The invention provides that whether a particular exception is dealt with by a secure mode exception handler executing in the secure mode or a non-secure mode exception handler executing in the non-secure mode is controlled in dependence upon a separately defined parameter.

Thus, by appropriate setting of these parameters, the hardware can be configured to direct different exceptions to different exception handlers as required by different operating systems or circumstances.

Some or all of the parameters controlling the selection of the exception handler may be stored in an exception trap mask register such that they can be changed under software control subject to the monitor kernal or other controlling program/mode having the required access permissions. As an alternative, some or all of the parameters may be defined by the hardware, such as being tied to particular signal values in the circut concerned once a design and implementation is settled.

The exception trap mask register may be conveniently provided in a configuration controlling coprocessor associated with the main processor and may be protected such that it is non-writable when the processor is in a non-secure mode.

The different exception handlers may be respectively provided by portions of a secure operating system and a non-secure operating system.

If the parameters specify that a switch between a secure mode and a non-secure mode is required, then this may be advantageously made via a monitor mode in which a high degree of protection may be given to the monitor program for controlling such security domain switching.

The monitor program may change the parameters controlling whether or not the exceptions should be handled by a secure exception handler or a non-secure exception handler.

In the context of a system having a monitor mode and a monitor program, the secure exception handler may be considered to be the monitor program itself and accordingly the parameters specify whether an exception should be handled in the existing security domain or whether it should be transferred to the monitor program in the monitor mode to be handled by the monitor program or as directed by the monitor program.

Viewed from another aspect the invention provides a method of processing data, said method comprising the steps of:

executing a program with a processor operable in a plurality of modes and either a secure domain or a non-secure domain including:

at least one secure mode being a mode in said secure domain; and

at least one non-secure mode being a mode in said non-secure domain; wherein

when said processor is executing a program in a secure mode said program has access to secure data which is not accessible when said processor is operating in a non-secure mode;

in response to one or more exception conditions triggering exception processing using an exception handler; wherein

said processor selects an exception handler in response to one or more parameters specifying which of said exceptions should be handled by a secure mode exception handler executing in a secure mode and which of said exceptions should be handled by an exception handler executing in a mode within a current one of said secure domain and said non-secure domain when that exception occurs.

The present invention will be described further, by way of example only, with reference to preferred embodiments thereof as illustrated in the accompanying drawings, in which:

Figure 1 is a block diagram schematically illustrating a data processing apparatus in accordance with preferred embodiments of the present invention;

Figure 2 schematically illustrates different programs operating in a non-secure domain and a secure domain;

Figure 3 schematically illustrates a matrix of processing modes associated with different security domains;

Figures 4 and 5 schematically illustrate different relationships between processing modes and security domains;

Figure 6 illustrates one programmer's model of a register bank of a processor depending upon the processing mode;

Figure 7 illustrates an example of providing separate register banks for a secure domain and a non-secure domain;

Figure 8 schematically illustrates a plurality of processing modes with switches between security domains being made via a separate monitor mode;

Figure 9 schematically illustrates a scenario for security domain switching using a mode switching software interrupt instruction;

Figure 10 schematically illustrates one example of how non-secure interrupt requests and secure interrupt requests may be processed by the system;

Figures 11A and 11B schematically illustrate an example of non-secure interrupt request processing and an example of secure interrupt request processing in accordance with Figure 10;

Figure 12 illustrates an alternative scheme for the handling of non-secure interrupt request signals and secure interrupt request signals compared to that illustrated in Figure 10;

Figures 13A and 13B illustrate example scenarios for dealing with a non-secure interrupt request and a secure interrupt request in accordance with the scheme illustrated in Figure 12;

Figure 14 is an example of a vector interrupt table;

Figure 15 schematically illustrates multiple vector interrupt tables associated with different security domains;

Figure 16 schematically illustrates an exception control register;

Figure 17 is a flow diagram illustrating how an instruction attempting to change a processing status register in a manner that alters the security domain setting can generate a separate mode change exception which in turn triggers entry into the monitor mode and running of the monitor program;

Figures 18 to 20 illustrate a view of different processing modes and scenario for switching between secure and non-secure domains in accordance with another example embodiment(s);

Figure 21 is a diagram illustrating the memory management logic used in one embodiment of the present invention to control access to memory;

Figure 22 is a block diagram illustrating the memory management logic of a second embodiment of the present invention used to control access to memory;

Figure 23 is a flow diagram illustrating the process performed in one embodiment of the present invention within the memory management logic to process a memory access request that specifies a virtual address;

Figure 24 is a flow diagram illustrating the process performed in one embodiment of the present invention within the memory management logic to process a memory access request that specifies a physical address;

Figure 25 schematically illustrates how the partition checker of preferred embodiments is operable to prevent access to a physical address within secure memory when the device issuing the memory access request is operating in a non-secure mode;

Figure 26 is a diagram illustrating the use of both a non-secure page table and a secure page table in preferred embodiments of the present invention;

Figure 27 is a diagram illustrating two forms of flag used within the main translation lookaside buffer (TLB) of preferred embodiments;

Figure 28 illustrates how memory may be partitioned after a boot stage in one embodiment of the present invention;

Figure 29 illustrates the mapping of the non-secure memory by the memory management unit following the performance of the boot partition in accordance with an embodiment of the present invention;

Figure 30 illustrates how the rights of a part of memory can be altered to allow a secure application to share memory with a non-secure application in accordance with an embodiment of the present invention;

Figure 31 illustrates how devices may be connected to the external bus of the data processing apparatus in accordance with one embodiment of the present invention;

Figure 32 is a block diagram illustrating how devices may be coupled to the external bus in accordance with the second embodiment of the present invention;

Figure 33 schematically shows possible granularity of monitoring functions for different modes and applications running on a processor;

Figure 34 shows possible ways of initiating different monitoring functions;

Figure 35 shows a table of control values for controlling availability of different monitoring functions;

Figure 36 shows a positive-edge triggered FLIP-FLOP view;

Figure 37 a scan chain cell;

Figure 38 shows a plurality of scan chain cells in a scan chain;

Figure 39 shows a debug TAP controller;

Figure 40A shows a debug TAP controller with a JADI input;

Figure 40B shows a scan chain cell with a bypass register;

Figure 41 schematically illustrates a processor comprising a core, scan chains and a Debug Status and Control Register;

Figure 42 schematically illustrates the factors controlling debug or trace initialisation;

Figures 43A and 43B show a summary of debug granularity;

Figure 44 schematically illustrates the granularity of debug while it is running;

Figure 45A and 45B show monitor debug when debug is enabled in secure world and when it is not enabled respectively.

Figure 1 is a block diagram illustrating a data processing apparatus in accordance with preferred embodiments of the present invention. The data processing apparatus incorporates a processor core 10 within which is provided an arithmetic logic unit (ALU) 16 arranged to execute sequences of instructions. Data required by the ALU 16 is stored within a register bank 14. The core 10 is provided with various monitoring functions to enable diagnostic data to be captured indicative of the activities of the processor core. As an example, an Embedded Trace Module (ETM) 22 is provided for producing a real time trace of certain activities of the processor core in dependence on the contents of certain control registers 26 within the ETM 22 defining which activities are to be traced. The trace signals are typically output to a trace buffer from where they can subsequently be analysed. A vectored interrupt controller 21 is provided for managing the servicing of a plurality of interrupts which may be raised by various peripherals ().

Further, as shown in Figure 1, another monitoring functionality that can be provided within the core 10 is a debug function, a debugging application external to the data processing apparatus being able to communicate with the core 10 via a Joint Test Access Group (JTAG) controller 18 which is coupled to one or more scan chains

12. Information about the status of various parts of the processor core 10 can be output via the scan chains 12 and the JTAG controller 18 to the external debugging application. An In Circuit Emulator (ICE) 20 is used to store within registers 24 conditions identifying when the debug functions should be started and stopped, and hence for example will be used to store breakpoints, watchpoints, etc.

The core 10 is coupled to a system bus 40 via memory management logic 30 which is arranged to manage memory access requests issued by the core 10 for access to locations in memory of the data processing apparatus. Certain parts of the memory may be embodied by memory units connected directly to the system bus 40, for example the Tightly Coupled Memory (TCM) 36, and the cache 38 illustrated in Figure 1. Additional devices may also be provided for accessing such memory, for example a Direct Memory Access (DMA) controller 32. Typically, various control registers 34 will be provided for defining certain control parameters of the various elements of the chip, these control registers also being referred to herein as coprocessor 15 (CP 15) registers.

The chip containing the core 10 will typically be coupled to an external bus 70 (for example a bus operating in accordance with the "Advanced Microcontroller Bus Architecture" (AMBA) specification developed by ARM Limited) via an external bus interface 42, and various devices may be connected to the external bus 70. These devices may include master devices such as a digital signal processor (DSP) 50, or a direct memory access (DMA) controller 52, as well as various slave devices such as the boot ROM 44, the screen driver 46, the external memory 56, an input/output (I/O) interface 60 or a key storage unit 64. These various slave devices illustrated in Figure 1 can all be considered as incorporating parts of the overall memory of the data processing apparatus. For example, the boot ROM 44 will form part of the addressable memory of the data processing apparatus, as will the external memory 56. Further, devices such as the screen driver 46, I/O interface 60 and key storage unit 64 will all include internal storage elements such as registers or buffers 48, 62, 66, respectively, which are all independently addressable as part of the overall memory of the data processing apparatus. As will be discussed in more detail later, a part of the memory, e.g. part of external memory 56, will be used to store one or more page tables 58 defining information relevant to control of memory accesses.

As will be appreciated by those skilled in the art, the external bus 70 will typically be provided with arbiter and decoder logic 54, the arbiter being used to arbitrate between multiple memory access requests issued by multiple master devices, for example the core 10, the DMA 32, the DSP 50, the DMA 52, etc, whilst the decoder will be used to determine which slave device on the external bus should handle any particular memory access request.

Figure 2 schematically illustrates various programs running on a processing system having a secure domain and a non-secure domain. The system is provided with a monitor program 72 which executes at least partially in a monitor mode. A security status flag is accessible only within the monitor mode and may be written by the monitor program 72. The monitor program 72 is responsible for managing all changes between the secure domain and the non-secure domain in either direction. From a view external to the core the monitor mode is always secure and the monitor program is in secure memory.

Within the non-secure domain there is provided a non-secure operating system 74 and a plurality of non-secure application programs 76, 78 which execute in co-operation with the non-secure operating system 74. In the secure domain, a secure kernel program 80 is provided. The secure kernel program 80 can be considered to form a secure operating system. Typically such a secure kernel program 80 will be designed to provide only those functions which are essential to processing activities which must be provided in the secure domain such that the secure kernel 80 can be as small and simple as possible since this will tend to make it more secure. A plurality of secure applications 82, 84 are illustrated as executing in combination with the secure kernel 80.

Figure 3 illustrates a matrix of processing modes associated with different security domains. In this particular example the processing modes are symmetrical with respect to the security domain and accordingly Mode 1 and Mode 2 exist in both secure and non-secure forms.

The monitor mode has the highest level of security access in the system and is the only mode entitled to switch the system between the non-secure domain and the secure domain in either direction. Thus, all domain switches take place via a switch to the monitor mode and the execution of the monitor program 72 within the monitor mode.

Figure 4 schematically illustrates another set of non-secure domain processing modes 1, 2, 3, 4 and secure domain processing modes a, b, c. In contrast to the symmetric arrangement of Figure 3, Figure 4 shows that some of the processing modes may not be present in one or other of the security domains. The monitor mode 86 is again illustrated as straddling the non-secure domain and the secure domain. The monitor mode 86 can be considered a secure processing mode, since the secure status flag may be changed in this mode and monitor program 72 in the monitor mode has the ability to itself set the security status flag it effectively provides the ultimate level of security within the system as a whole.

Figure 5 schematically illustrates another arrangement of processing modes with respect to security domains. In this arrangement both secure and non-secure domains are identified as well as a further domain. This further domain may be such that it is isolated from other parts of a system in a way that it does not need to interact with either of the secure domain or non-secure domain illustrated and as such the issue of to which of these it belongs to is not relevant.

As will be appreciated a processing system, such as a microprocessor is normally provided with a register bank 88 in which operand values may be stored. Figure 6 illustrates a programmer's model view of an example register bank with dedicated registers being provided for certain of the register numbers in certain of the processing modes. More particularly, the example of Figure 6 is an extension of the known ARM register bank (e.g. as provided in ARM7 processors of ARM Limited, Cambridge, England) which is provided with a dedicated saved program status register, a dedicated stack pointer register and a dedicated link register R14 for each processing mode, but in this case extended by the provision of a monitor mode. As illustrated in Figure 6, the fast interrupt mode has additional dedicated registers provided such that upon entry of the fast interrupt mode there is no need to save and

then restore register contents from other modes. The monitor mode may in alternative embodiments also be provided with dedicated further registers in a similar manner to the fast interrupt mode so as to speed up processing of a security domain switch and reduce system latency associated with such switches.

Figure 7 schematically illustrates another embodiment in which the register bank 88 is provided in the form of two complete and separate register banks that are respectively used in the secure domain and the non-secure domain. This is one way in which secure data stored within registers operable in the secure domain can be prevented from becoming accessible when a switch is made to the non-secure domain. However, this arrangement hinders the possibility of passing data from the non-secure domain to the secure domain as may be permitted and desirable by using the fast and efficient mechanism of placing it in a register which is accessible in both the non-secure domain and the secure domain.

An important advantage of having secure register bank is to avoid the need for flushing the contents of registers before switching from one world to the other. If latency is not a critical issue, a simpler hardware system with no duplicated registers for the secure domain world may be used, e.g. Figure 6. The monitor mode is responsible switching from one domain to the other. Restoring context, saving previous context, as well as flushing registers is performed by a monitor program at least partially executing in monitor mode. The system behaves thus like a virtualisation model. This type of embodiment is discussed further below. Reference should be made to, for example, the programmer's model of the ARM7 upon which the security features described herein build.

Processor Modes

Instead of duplicating modes in secure world, the same modes support both secure and non-secure domains (see Figure 8). Monitor mode is aware of the current status of the core, either secure or non-secure (e.g. as read from an S bit stored is a coprocessor configuration register).

In the Figure 8, whenever an SMI (Software Monitor Interrupt instruction) occurs, the core enters monitor mode to switch properly from one world to the other.

With reference to Figure 9:

1. The scheduler launches thread 1
2. Thread 1 needs to perform a secure function => SMI secure call, the core enters monitor mode. Under hardware control the current PC and CPSR (current processor status register) are stored in R14_mon and SPSR_mon (saved processor status register for the monitor mode) and IRQ/FIQ interrupts are disabled.
3. The monitor program does the following tasks:
   - The S bit is set (the secure status flag).
   - Saves at least R14_mon and SPSR_mon in a stack so that non-secure context cannot be lost if an exception occurs whilst the secure application is running.
   - Checks there is a new thread to launch: secure thread 1. A mechanism (via thread ID table) indicates that thread 1 is active in the secure world.
   - IRQ/FIQ interrupts are re-enabled. A secure application can then start in secure user mode.
4. Secure thread 1 runs until it finishes, then branches (SMI) onto the 'return from secure' function of the monitor program mode (IRQ/FIQ interrupts are then disabled when the core enters monitor mode)
5. The 'return from secure' function does the following tasks:
   - indicates that secure thread 1 is finished (e.g., in the case of a thread ID table, remove thread 1 from the table).
   - Restore from stack non-secure context and flush required registers, so that no secure data can be read once return has been made to the non-secure domain.
   - Then branches back to the non-secure domain with a SUBS instruction (this restores the program counter to the correct point and updates the status flags), restoring the PC (from restored R14_mon) and CPSR (from SPSR_mon). So, the return point in the non-secure domain is the instruction following the previously executed SMI in thread 1.
6. Thread 1 executes until the end, then gives the hand back to the scheduler.

In other embodiments it may be desired not to allow SMIs to occur in user modes.

Secure World Entry

Reset

When a hardware reset occurs, the MMU is disabled and the ARM core (processor) branches to secure supervisor mode with the S bit set. Once the secure boot is terminated an SMI to go to monitor mode may be executed and the monitor can switch to the OS in non-secure world (non-secure svc mode) if desired. If it is desired to use a legacy OS this can simply boo in secure supervisor mode and ignore the secure state.

SMI INSTRUCTION

This instruction (a mode switching software interrupt instruction) can be called from any non-secure modes in the non-secure domain (as previously mentioned it may be desired to restrict SMIs to privileged modes), but the target entry point determined by the associated vector is always fixed and within monitor mode. Its up to the SMI handler to branch to the proper secure function that must be run (e.g. controlled by an operand passed with the instruction).

Passing parameters from non-secure world to secure world can be performed using the shared registers of the register bank within a Figure 6 type register bank.

When a SMI occurs in non-secure world, the ARM core may do the following actions in hardware:

- Branch to SMI vector (in secure memory access is allowed since you will now be in monitor mode) into monitor mode
- Save PC into R14_mon and CPSR into SPSR_mon
- Set the S bit using the monitor program
- Start to execute secure exception handler in monitor mode (restore/save context in case of multi-threads)

- Branch to secure user mode (or another mode, like svc mode) to execute the appropriate function
- IRQ and FIQ are disabled while the core is in monitor mode (latency is increased)

Secure World Exit

There are two possibilities to exit secure world:

- The secure function is finished and we return into previous non-secure mode that had called this function.
- The secure function is interrupted by a non-secure exception (e.g. IRQ/FIQ/SMI).

Normal End of Secure Function

The secure function terminates normally and we need to resume an application in the non-secure world at the instruction just after the SMI. In the secure user mode, a 'SMI' instruction is performed to return to monitor mode with the appropriate parameters corresponding to a 'return from secure world' routine. At this stage, the registers are flushed to avoid leakage of data between non-secure and secure worlds, then non-secure context general purpose registers are restored and non-secure banked registers are updated with the value they had in non-secure world. R14_mon and SPSR_mon thus get the appropriate values to resume the non-secure application after the SMI, by executing a 'MOVS PC, R14' instruction.

Exit of Secure Function Due to a Non-Secure Exception

In this case, the secure function is not finished and the secure context must be saved before going into the non-secure exception handler, whatever the interrupts are that need to be handled.

Secure Interrupts

There are several possibilities for secure interrupts.

Two possible solutions are proposed which depend on:

- What kind of interrupt it is (secure or non-secure)
- What mode the core is in when the IRQ occurs (either in secure or in non-secure world)

Solution One

In this solution, two distinct pins are required to support secure and non-secure interrupts.

While in Non Secure world, if

- an IRQ occurs, the core goes to IRQ mode to handle this interrupt as in ARM cores such as the ARM7
- a SIRQ occurs, the core goes to monitor mode to save non-secure context and then to a secure IRQ handler to deal with the secure interrupt.

While in Secure world, if

- an SIRQ occurs, the core goes to the secure IRQ handler. The core does not leave the secure world
- an IRQ occurs, the core goes to monitor mode where secure context is saved, then to a non-secure IRQ handler to deal with this non-secure interrupt.

In other words, when an interrupt that does not belong to the current world occurs, the core goes directly to monitor mode, otherwise it stays in the current world (see Figure 10).

IRQ Occurring in Secure World

See Figure 11A:

1. The scheduler launches thread 1.
2. Thread 1 needs to perform a secure function => SMI secure call, the core enters monitor mode. Current PC and CPSR are stored in R14_mon and SPSR_mon, IRQ/FIQ are disabled.

3. The monitor handler (program) does the following tasks:

- The S bit is set.
- Saves at least R14_mon and SPSR_mon in a stack (and possibly other registers are also pushed) so that non-secure context cannot be lost if an exception occurs whilst the secure application is running.
- Checks there is a new thread to launch: secure thread 1. A mechanism (via thread ID table) indicates that thread 1 is active in the secure world.
- Secure application can then start in the secure user mode. IRQ/FIQ are then re-enabled.

4. An IRQ occurs while secure thread 1 is running. The core jumps directly to monitor mode (specific vector) and stores current PC in R14_mon and CPSR in SPSR_mon in monitor mode, (IRQ/FIQ are then disabled).

5. Secure context must be saved, previous non-secure context is restored. The monitor handler may be to IRQ mode to update R14_irq/SPSR_irq with appropriate values and then passes control to a non-secure IRQ handler.

6. The IRQ handler services the IRQ, then gives control back to thread 1 in the non-secure world. By restoring SPRS_irq and R14_irq into the CPSR and PC, thread 1 is now pointing onto the SMI instruction that has been interrupted.

7. The SMI instruction is re-executed (same instruction as 2).

8. The monitor handler sees this thread has previously been interrupted, and restores the thread 1 context. It then branches to secure thread 1 in user mode, pointing onto the instruction that has been interrupted.

9. Secure thread 1 runs until it finishes, then branches onto the 'return from secure' function in monitor mode (dedicated SMI).

10. The 'return from secure' function does the following tasks:

- indicates that secure thread 1 is finished (i.e., in the case of a thread ID table, remove thread 1 from the table).
- restore from stack non-secure context and flush required registers, so that no secure data can be read once a return is made to non-secure world.
- branches back to the non-secure world with a SUBS instruction, restoring the PC (from restored R14_mon) and CPSR (from SPSR_mon). So, the return point in the non-secure world should be the instruction following the previously executed SMI in thread 1.

11. Thread 1 executes until the end, then gives control back to the scheduler.

SIRQ Occurring in Non-Secure World

See Figure 11B:

1. The schedule launches thread 1
2. A SIRQ occurs while secure thread 1 is running. The core jumps directly to monitor mode (specific vector) and stores current PC in R14_mon and CPSR in SPSR_mon in monitor mode, IRQ/FIQ are then disabled.
3. Non-Secure context must be saved, then the core goes to a secure IRQ handler.
4. The IRQ handler services the SIRQ, then gives control back to the monitor mode handler using an SMI with appropriate parameters.
5. The monitor handler restores non-secure context so that a SUBS instruction makes the core return to the non-secure world and resumes the interrupted thread 1.
6. Thread 1 executes until the end, then gives the hand back to the scheduler.

The mechanism of Figure 11A has the advantage of providing a deterministic way to enter secure world. However, there are some problems associated with interrupt priority: e.g. while a SIRQ is running in secure interrupt handler, a non-secure IRQ with higher priority may occur. Once the non-secure IRQ is finished, there is a need to recreate the SIRQ event so that the core can resume the secure interrupt.

Solution Two

In this mechanism (See Figure 12) two distinct pins, or only one, may support secure and non-secure interrupts. Having two pins reduces interrupt latency.

While in Non Secure world, if

- an IRQ occurs, the core goes to IRQ mode to handle this interrupt like in ARM7 systems
- a SIRQ occurs, the core goes to an IRQ handler where an SMI instruction will make the core branch to monitor mode to save non-secure context and then to a secure IRQ handler to deal with the secure interrupt.

While in a Secure world, if

- a SIRQ occurs, the core goes to the secure IRQ handler. The core does not leave the secure world
- an IRQ occurs, the core goes to the secure IRQ handler where an SMI instruction will make the core branch to monitor mode (where secure context is saved), then to a non-secure IRQ handler to deal with this non-secure interrupt.

IRQ Occurring In Secure World

See Figure 13A:

1. The schedule launches thread 1.
2. Thread 1 needs to perform a secure function => SMI secure call, the core enters monitor mode. Current PC and CPSR are stored in R14_mon and SPSR_mon, IRQ/FIQ are disabled.
3. The monitor handler does the following tasks:
   - The S bit is set.
   - Saves at least R14_mon and SPSR_mon in a stack (eventually other registers) so that non-secure context cannot be lost if an exception occurs whilst the secure application is running.
   - Checks there is a new thread to launch: secure thread 1. A mechanism (via thread ID table) indicates that thread 1 is active in the secure world.
   - Secure application can then start in the secure user mode. IRQ/FIQ are re-enabled.
4. An IRQ occurs while secure thread 1 is running. The core jumps directly to secure IRQ mode.
5. The core stores current PC in R14_irq and CPSR in SPSR_irq. The IRQ handler detects this is a non-secure interrupt and performs a SMI to enter monitor mode with appropriate parameters.
6. Secure context must be saved, previous non-secure context is restored. The monitor handler knows where the SMI came from by reading the CPSR. It can also go to IRQ mode to read R14_irq/SPSR_irq to save properly secure context. It can

also save in these same registers the non-secure context that must be restored once the IRQ routine will be finished.

7. The IRQ handler services the IRQ, then gives control back to thread 1 in the non-secure world. By restoring SPRS_irq and R14_irq into the CPSR and PC, the core is now pointing onto the SMI instruction that has been interrupted.

8. The SMI instruction is re-executed (same instruction as 2).

9. The monitor handler sees this thread has previously been interrupted, and restores the thread 1 context. It then branches to secure thread 1 in user mode, pointing to the instruction that has been interrupted.

10. Secure thread 1 runs until it finishes, then branches onto the 'return from secure'; function in monitor mode (dedicated SMI).

11. The 'return from secure' function does the following tasks:

- indicates that secure thread 1 is finished (i.e., in the case of a thread ID table, remove thread 1 from the table).
- restores from stack non-secure context and flushes required registers, so that no secure information can be read once we return in non-secure world.
- branches back to the non-secure world with a SUBS instruction, restoring the PC (from restored R14_mon) and CPSR (from SPSR_mon). The return point in the non-secure world should be the instruction following the previously executed SMI in thread 1.

11. Thread 1 executes until the end, then gives the hand back to the scheduler.

SIRQ Occurring in Non-Secure World

See Figure 13B:

1. The schedule launches thread 1.
2. A SIRQ occurs while secure thread 1 is running.
3. The core jumps directly irq mode and stores current PC in R14_irq and CPSR in SPSR_irq. IRQ is then disabled. The IRQ handler detects this is a SIRQ and performs a SMI instruction with appropriate parameters.
4. Once in monitor mode, non-secure context must be saved, then the core goes to a secure IRQ handler.

5. The secure IRQ handler services the SIRQ service routine, then gives control back to monitor with SMI with appropriate parameters.

6. The monitor handler restores non-secure context so that a SUBS instruction makes the core returns to non-secure world and resumes the interrupted IRQ handler.

7. The IRQ handler may then return to the non-secure thread by performing a SUBS.

8. Thread 1 executes until the end, then gives control back to the scheduler.

With the mechanism of Figure 12, there is no need to recreate the SIRQ event in the case of nested interrupts, but there is no guarantee that secure interrupts will be performed.

Exception Vectors

At least two physical vector tables are kept (although from a virtual address point of view they may appear as a single vector table), one for the non-secure world in non-secure memory, the one for the secure world in secure memory (not accessible from non-secure world). The different virtual to physical memory mappings used in the secure and non-secure worlds effectively allow the same virtual memory addresses to access different vector tables stored in physical memory. The monitor mode may always use flat memory mapping to provide a third vector table in physical memory.

If the interrupts follow the Figure 12 mechanism, there would be the following vectors shown in Figure 14 for each table. This vector set is duplicated in both secure and non-secure memory.

Exception	Vector Offset	Corresponding Mode
Reset	0x00	Supervisor Mode (S bit set)
Undef	0x04	Monitor mode/Undef mode
SWI	0x08	Supervisor mode/Monitor mode

Prefetch Abort	0x0C	Abort mode/Monitor mode
Data Abort	0x10	Abort mode/Monitor Mode
SMI	0x14	Undef mode/Monitor mode
IRQ/SIRQ	0x18	IRQ mode
FIQ	0x1C	FIQ mode

NB. The Reset entry is only in the secure vector table. When a Reset is performed in non secure world, the core hardware forces entry of supervisor mode and setting of the S bit so that the Reset vector can be accessed in secure memory.

Figure 15 illustrates three exception vector tables respectively applicable to a secure mode, a non–secure mode and the monitor mode. These exception vector tables may be programmed with exception vectors in order to match the requirements and characteristics of the secure and non-secure operating systems. Each of the exception vector tables may have an associated vector table base address register within CP15 storing a base address pointing to that table within memory. When an exception occurs the hardware will reference the vector table base address register corresponding to the current state of the system to determine the base address of the vector table to be used. Alternatively, the different virtual to physical memory mappings applied in the different modes may be used to separate the three different vector table sstored at different physical memory addresses. As illustrated in Figure 16, an exception trap mask register is provided in a system (configuration controlling) coprocessor (CP15) associated with the processor core. This exception trap mask register provides flags associated with respective exception types. These flags indicate whether the hardware should operate to direct processing to either the vector for the exception concerned within its current domain or should force a switch to the monitor mode (which is a type of secure mode) and then follow the vector in the monitor mode vector table. The exception trap mask register (exception control register) is only writable from the monitor mode. It may be that read access is also prevented to the exception trap mask register when in a non-secure mode. It will be

seen that the exception trap mask register of Figure 16 does not include a flag for the reset vector as the system is configured to always force this to jump to the reset vector in the secure supervisor mode as specified in the secure vector table in order to ensure a secure boot and backwards compatibility. It will be seen that in Figure 15, for the sake of completeness, reset vectors have been shown in the vector tables other than the secure supervisor mode secure vector table.

Figure 16 also illustrates that the flags for the different exception types within the exception trap mask register are programmable, such as by the monitor program during secure boot. Alternatively, some or all of the flags may in certain implementations be provided by physical input signals, e.g. the secure interrupt flag SIRQ may be hardwired to always force monitor mode entry and execution of the corresponding monitor mode secure interrupt request vector when a secure interrupt signal is received. Figure 16 illustrates only that portion of the exception trap register concerned with non-secure domain exceptions, a similar set of programmable bits will be provided for secure domain exceptions.

Whilst it will be understood from the above that at one level the hardware acts to either force an interrupt to be serviced by the current domain exception handler or the monitor mode exception handler depending upon the exception control register flags, this is only the first level of control that is applied. As an example, it is possible for an exception to occur in the secure mode, the secure mode exception vector to be followed to the secure mode exception handler, but this secure mode exception handler then decide that the exception is of a nature that it is better dealt with by the non-secure exception handler and accordingly utilise an SMI instruction to switch to the non-secure mode and invoke the non-secure exception handler. The converse is also possible where the hardware might act to initiate the non-secure exception handler, but this then execute instructions which direct processing to the secure exception handler or the monitor mode exception handler.

Figure 17 is a flow diagram schematically illustrating the operation of the system so as to support another possible type of switching request associated with a

new type of exception. At step 98 the hardware detects any instruction which is attempting to change to monitor mode as indicate in a current program status register (CPSR). When such an attempt is detected, then a new type of exception is triggered, this being referred to herein as a CPSR violation exception. The generation of this CPSR violation exception at step 100 results in reference to an appropriate exception vector within the monitor mode and the monitor program is run at step 102 to handle the CPSR violation exception.

It will be appreciated that the mechanisms for initiating a switch between secure domain and non-secure domain discussed in relation to Figure 17 may be provided in addition to support for the SMI instruction previously discussed. This exception mechanism may be provided to respond to unauthorised attempts to switch mode as all authorised attempts should be made via an SMI instruction. Alternatively, such a mechanism may be legitimate ways to switch between the secure domain and the non-secure domain or may be provided in order to give backwards compatibility with existing code which, for example, might seek to clear the processing status register as part of its normal operation even though it was not truly trying to make an unauthorised attempt to switch between the secure domain and the non-secure domain.

A description of an alternative embodiment(s) of the present technique considered from a programmer's model view is given below in relation to Figures 18 to 20 as follows:

In the following description, we will use the following terms that must be understood in the context of an ARM processor as designed by ARM Limited, of Cambridge, England.

- S bit : Secure state bit, contained in a dedicated CP15 register.
- 'Secure/Non-Secure state'. This state is defined by the S bit value. It indicates whether the core may access the Secure world (when it is in Secure state, i.e. S=1) or is restricted to the Non-secure world only (S=0). Note that the Monitor mode (see further) overrides the S bit status.
- 'Non-Secure World' groups all hardware/software accessible to non-secure applications that do not require security.

- 'Secure World' groups all hardware/software (core, memory...) that is only accessible when we execute secure code.
- Monitor mode: new mode that is responsible for switching the core between the Secure and Non-secure state.

As a brief summary

- The core can always access the Non-secure world.
- The core can access the Secure world only when it is in Secure state or Monitor mode.

- SMI: Software Monitor Interrupt: New instruction that will make the core enter the Monitor mode through a dedicated SMI exception vector.'Thread ID': is the identifier associated to each thread (controlled by an OS). For some types of OS where the OS runs in non-secure world, each time a secure function is called, it will be necessary to pass as a parameter the current thread ID to link the secure function to its calling non-secure application. The secure world can thus support multi-threads.
- Secure Interrupt defines an interrupt generated by a Secure peripheral.

Programmer's model

Carbon Core Overview

The concept of the Carbon architecture, which is the term used herein for processors using the present techniques, consists in separating two worlds, one secure and one non-secure. The secure world must not leak any data to non-secure world.

In the proposed solution, the secure and non-secure states will share the same (existing) register bank. As a consequence, all current modes present in ARM cores (Abort, Undef, Irq, User, ...) will exist in each state.

The core will know it operates in secure or non-secure state thanks to a new state bit, the S (secure) bit, instantiated in a dedicated CP15 register.

Controlling which instruction or event is allowed to modify the S bit, i.e. to change from one state to the other, is a key feature of the security of the system. The current solution proposes to add a new mode, the Monitor mode, that will "supervise" switching between the two states. The Monitor mode, by writing to the appropriate CP15 register, would be the only one allowed to alter the S bit.

Finally, we propose to add some flexibility to the exception handling. All exceptions, apart from the reset, would be handled either in the state where they happened, or would be directed to the Monitor mode. This would be left configurable thanks to a dedicated CP15 register.

The details of this solution are discussed in the following paragraphs.

Processor state and modes

Carbon new features

Secure or Non-secure state (S bit)

One major feature of the Carbon core is the presence of the S bit, which indicates whether the core is in a Secure (S=1) or Non-secure (S=0) state. When in Secure state, the core would be able to access any data in the Secure or Non-secure worlds. When in Non-Secure state, the core would be restricted to the Non-secure world only.

The only exception to this rule concerns the Monitor mode, which overrides the S bit information. Even when S=0, the core will perform Secure privileged accesses when it is in Monitor mode. See next paragraph, Monitor mode, for further information

The S bit can only be read and written in Monitor mode. Whatever the S bit value, if any other mode tries to access it, this will be either ignored or result in an Undefined exception.

All exceptions, apart from Reset, have no effect on the Secure state bit. On Reset, the S bit will be set, and the core will start in Supervisor mode. Refer to the boot section for detailed information.

Secure/Nonsecure states are separate and operate independently of the ARM/Thumb/Java states.

Monitor mode

One other important feature of the Carbon system is the creation of a new mode, the Monitor mode. This will be used to control the core switching between the Secure and Non-secure states . It will always be considered as a secure mode, i.e. whatever the value of the S bit, the core will always perform Secure Privileged accesses to the external world when it is in Monitor mode.

Any Secure privileged mode (i.e. privileged modes when S=1) would be able to switch to Monitor mode by simply writing the CPSR mode bits (MSR, MOVS, or equivalent instruction). However, this would be forbidden in any Non-secure mode or Secure user mode. If this ever happens, the instruction would be ignored or cause an exception.

There may be a need for a dedicated CPSR violation exception. This exception would be raised by any attempt to switch to Monitor mode by directly writing the CPSR from any Non-secure mode or Secure user mode.

All exceptions except Reset are in effect disabled when Monitor mode is active:

- all interrupts are masked;
- all memory exceptions are either ignored or cause a fatal exception.
- undefined/SWI/SMI are ignored or cause a fatal exception.

When entering Monitor mode, the interrupts are automatically disabled and the system monitor should be written such that none of the other types of exception can happen while the system monitor is running.

Monitor mode needs to have some private registers. This solution proposes that we only duplicate the minimal set of registers, i.e R13 (sp_mon), R14 (ir_mon) and SPSR (spsr_mon).

In Monitor mode, the MMU will be disabled (flat address map) as well as the MPU or partition checker (the Monitor mode will always perform secure privileged external accesses). However, specially programmed MPU region attributes (cacheability, ...) would still be active.

New instruction

This proposal requires adding one new instruction to the existing ARM instruction set.

The SMI (Software Monitor Interrupt) instruction would be used to enter the Monitor mode, branching at a fixed SMI exception vector. This instruction would be mainly used to indicate to the Monitor to swap between the Non-secure and Secure State.

As an alternative (or in addition) it would be possible to add a new instruction to allow the Monitor mode to save/restore the state of any other mode onto/from the Monitor stack to improve context switching performance.

Processor Modes

As discussed in the previous paragraph, only one new mode is added in the core, the Monitor mode. All existing modes remain available, and will exist both in the secure and non-secure states.

In fact, Carbon users will see the structure illustrated in Figure 18.

Processor registers

This embodiment proposes that the secure and the non-secure worlds share the same register bank. This implies that, when switching from one world to the other through the Monitor mode, the system monitor will need to save the first world context, and create (or restore) a context in the second world.

Passing parameters becomes an easy task: any data contained in a register in the first world will be available in the same register in the second world once the system monitor has switched the S bit.

However, apart from a limited number of registers dedicated to passing parameters, which will need to be strictly controlled, all other registers will need to be flushed when passing from Secure to Non-secure state in order to avoid any leak of Secure data. This will need to be ensured by the Monitor kernel.

The possibility of implementing a hardware mechanism or a new instruction to directly flush the registers when switching from Secure to Non-secure state is also a possibility.

Another solution proposed involves duplicating all (or most of) the existing register bank, thus having two physically separated register banks between the Secure and Non-secure state. This solution has the main advantage of clearly separating the secure and non-secure data contained in the registers. It also allows fast context switching between the secure and non-secure states. However, the drawback is that passing parameters through registers becomes difficult, unless we create some dedicated instructions to allow the secure world access the non-secure registers

Figure 19 illustrates the available registers depending on the processor mode. Note that the processor state has no impact on this topic.

Exceptions

Secure interrupts

Current Solution

It is currently proposed to keep the same interrupt pins as in the current cores, i.e. IRQ and FIQ. In association with the Exception Trap Mask register (defined later in the document), there should be sufficient flexibility for any system to implement and handle different kind of interrupts.

VIC enhancement

We could enhance the VIC (Vectored Interrupt Controller) in the following way: the VIC may contain one Secure information bit associated to each vectored address. This bit would be programmable by the Monitor or Secure privileged modes only. It would indicate whether the considered interrupt should be treated as Secure, and thus should be handled on the Secure side.

We would also add two new Vector Address registers, one for all Secure Interrupts happening in Non-Secure state, the other one for all Non-Secure interrupts happening in Secure state.

The S bit information contained in CP15 would be also available to the VIC as a new VIC input.

The following table summarizes the different possible scenarios, depending on the status of the incoming interrupt (Secure or Non-secure, indicated by the S bit associated to each interrupt line) and the state of the core (S bit in CP15 = S input signal on the VIC).

	Core in secure state (CP15 – S=1)	**Core in Non-secure state (CP15 - S=0)**
Secure Interrupt	No need to switch between worlds. The VIC directly presents to the core the Secure address associated to the interrupt line. The core simply has to branch at this address where it should find the associated ISR.	The VIC has no Vector associated to this interrupt in the Non-secure domain. It thus presents to the core the address contained in the Vector address register dedicated to all Secure interrupts occurring in Non-secure world. The core, still in Non-secure world, then branches to this address, where it should find an SMI instruction to switch to Secure world. Once in Secure world, it would be able to have access to the correct ISR.

Non-Secure Interrupt	The VIC has no Vector associated to this interrupt in the Secure domain. It thus presents to the core the address contained in the Vector address register dedicated to all Non-secure interrupts occurring in Secure world. The core, still in Secure world, then branches to this address, where it should find an SMI instruction to switch to Non-secure world. Once in Non-secure world, it would be able to have access to the correct ISR.	No need to switch between worlds. The VIC directly presents to the core the Non-secure address associated to the interrupt line. The core simply has to branch at this address where it should find the associated Non-secure ISR.

Exception handling configurability

In order to improve Carbon flexibility, a new register, the Exception Trap Mask, would be added in CP15. This register would contain the following bits:

- Bit 0: Undef exception (Non-secure state)
- Bit 1: SWI exception (Non-secure state)
- Bit 2: Prefetch abort exception (Non-secure state)
- Bit 3: Data abort exception (Non-secure state)
- Bit 4: IRQ exception (Non-secure state)
- Bit 5: FIQ exception (Non-secure state)
- Bit 6: SMI exception (both Non-secure/Secure states)
- Bit 16: Undef exception (Secure state)
- Bit 17: SWI exception (Secure state)
- Bit 18: Prefetch abort exception (Secure state)
- Bit 19: Data abort exception (Secure state)
- Bit 20: IRQ exception (Secure state)
- Bit 21: FIQ exception (Secure state)

The Reset exception does not have any corresponding bit in this register. Reset will always cause the core to enter the Secure supervisor mode through its dedicated vector.

If the bit is set, the corresponding exception makes the core enter the Monitor mode. Otherwise, the exception will be handled in its corresponding handler in the world where it occurred.

This register would only be visible in Monitor mode. Any instruction trying to access it in any other mode would be ignored.

This register should be initialized to a system-specific value, depending upon whether the system supports a monitor or not. This functionality could be controlled by a VIC.

Exception vectors tables

As there will be separate Secure and Non-secure worlds, we will also need separate Secure and Non-secure exception vectors tables.

Moreover, as the Monitor can also trap some exceptions, we may also need a third exception vectors table dedicated to the Monitor.

The following table summarizes those three different exception vectors tables:

In non-secure memory:

Address	Exception	Mode	Automatically accessed when
0x00	-	-	
0x04	Undef	Undef	Undefined instruction executed when core is in Non-Secure state and Exception Trap Mask reg [Non-secure Undef]=0

0x08	SWI	Supervisor	SWI instruction executed when core is in Non-Secure state and Exception Trap Mask reg [Non-secure SWI]=0
0x0C	Prefetch Abort	Abort	Aborted instruction when core is in Non-Secure state and Exception Trap Mask reg [Non-secure PAbort]=0
0x10	Data Abort	Abort	Aborted data when core is in Non-Secure state and Exception Trap Mask reg [Non-secure DAbort]=0
0x14	Reserved		
0x18	IRQ	IRQ	IRQ pin asserted when core is in Non-Secure state and Exception Trap Mask reg [Non-secure IRQ]=0
0x1C	FIQ	FIQ	FIQ pin asserted when core is in Non-Secure state and Exception Trap Mask reg [Non-secure FIQ]=0

In secure memory:

Address	**Exception**	**Mode**	**Automatically accessed when**
0x00	Reset*	Supervisor	Reset pin asserted
0x04	Undef	Undef	Undefined instruction executed when core is in Secure state and Exception Trap Mask reg [Secure Undef]=0
0x08	SWI	Supervisor	SWI instruction executed when core is in Secure state and Exception Trap Mask reg [Secure SWI]=0
0x0C	Prefetch Abort	Abort	Aborted instruction when core is in Secure state and Exception Trap Mask reg [Secure PAbort]=0
0x10	Data Abort	Abort	Aborted data when core is in Secure state and Exception Trap Mask reg [Secure Dabort]=0
0x14	Reserved		

0x18	IRQ	IRQ	IRQ pin asserted when core is in Secure state and Exception Trap Mask reg [Secure IRQ]=0
0x1C	FIQ	FIQ	FIQ pin asserted when core is in Secure state and Exception Trap Mask reg [Secure FIQ]=0

* Refer to "Boot" section for further explanation on the Reset mechanism

In Monitor memory (flat mapping):

Address	**Exception**	**Mode**	**Automatically accessed when**
0x00	-	-	-
0x04	Undef	Monitor	Undefined instruction executed when core is in Secure state and Exception Trap Mask reg [Secure Undef]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure Undef]=1
0x08	SWI	Monitor	SWI instruction executed when core is in Secure state and Exception Trap Mask reg [Secure SWI]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure SWI]=1
0x0C	Prefetch Abort	Monitor	Aborted instruction when core is in Secure state and Exception Trap Mask reg [Secure IAbort]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure Iabort]=1
0x10	Data Abort	Monitor	Aborted data when core is in Secure state and Exception Trap Mask reg [Secure PAbort]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure Pabort]=1
0x14	SMI	Monitor	

0x18	IRQ	Monitor	- IRQ pin asserted when core is in Secure state and Exception Trap Mask reg [Secure IRQ]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure IRQ]=1
0x1C	FIQ	Monitor	- FIQ pin asserted when core is in Secure state and Exception Trap Mask reg [Secure FIQ]=1 core is in Non-secure state and Exception Trap Mask reg [Non-secure FIQ]=1

In Monitor mode, the exceptions vectors may be duplicated, so that each exception will have two different associated vector:

- One for the exception arising in Non-secure state
- One for the exception arising in Secure state

This may be useful to reduce the exception latency, because the monitor kernel does not have any more the need to detect the originating state where the exception occurred.

Note that this feature may be limited to a few exceptions, the SMI being one of the most suitable candidates to improve the switching between the Secure and Non-secure states.

Switching between worlds

When switching between states, the Monitor mode must save the context of the first state on its Monitor stack, and restore the second state context from the Monitor stack.

The Monitor mode thus needs to have access to any register of any other modes, including the private registers (r14, SPSR, ..).

To handle this, the proposed solution consists in giving any privilege mode in Secure state the rights to directly switch to Monitor mode by simply writing the CPSR.

With such a system, switching between worlds is performed as follows:

- enter Monitor mode
- set the S bit
- switch to supervisor mode - save the supervisor registers on the MONITOR stack (of course the supervisor mode will need to have access to the Monitor stack pointer, but this can be easily done, for example by using a common register (R0 to R8))
- switch to System mode - save the registers (=same as the user mode) on the Monitor stack
- IRQ registers on the Monitor stack

etc ... for all modes

- Once all private registers of all modes are saved, revert to Monitor mode with a simple MSR instruction (= simply write Monitor value in the CPSR mode field)

The other solutions have also been considered:

- Add a new instruction that would allow the Monitor to save other modes'private registers on its own stack.
- Implement the Monitor as a new "state", i.e. being able to be in Monitor state (to have the appropriate access rights) and in IRQ (or any other) mode, to see the IRQ (or any other) private registers.

Boot mechanism

The boot mechanism must respect the following features:

- Keep compatibility with legacy OSes.
- Boot in most privileged mode to ensure the security of the system.

As a consequence, Carbon cores will boot in Secure Supervisor mode.

The different systems will then be:

- For systems wanting to run legacy OSes, the S bit is not taken into account and the core will just see it boots in Supervisor mode.
- For systems wanting to use the Carbon features, the core boots in Secure privileged mode which should be able to configure all secure protections in the system (potentially after swapping to Monitor mode)

Basic Scenario (See Figure 20)

1. Thread 1 is running in non-secure world (S bit = 0)
   This thread needs to perform a secure function => SMI instruction.
2. The SMI instruction makes the core enter the Monitor mode through a dedicated non-secure SMI vector.
   LR_mon and SPSR_mon are used to save the PC and CPSR of the non secure mode.
   The S bit remains unchanged (i.e. non-secure state).
   The monitor kernel saves the non-secure context on the monitor.
   It also pushes LR_mon and SPSR_mon.
   The monitor kernel then changes the "S" bit by writing into the CP15 register.
   It must keep track that a "secure thread 1" will be started in the secure world (e.g. by updating a Thread ID table).
   Finally, it exits the monitor mode and switches to secure supervisor mode (MOVS instruction after having updated LR_mon and SPSR_mon?).
3. The secure kernel dispatches the application to the right secure memory location, then switches to user mode (e.g. using a MOVS).
4. The secure function in executed in secure user mode. Once finished, it calls an "exit" function by performing an appropriate SWI.
5. The SWI instruction makes the core enter the secure svc mode through a dedicated SWI vector, that in turn performs the "exit" function. This "exit" function ends with an "SMI" to switch back to monitor mode.
6. The SMI instruction makes the core enter the monitor mode through a dedicated secure SMI vector.
   LR_mon and SPSR_mon are used to save the PC and CPSR of the Secure svc mode.
   The S bit remains unchanged (i.e. Secure State).

The monitor kernel registers the fact that secure thread 1 is finished (removes the secure thread 1 ID from the thread ID table?).
It then changes the "S" bit by writing into the CP15 register, returning to non-secure state.
The monitor kernel restores the non-secure context from the monitor stack.
It also load the LR_mon and CPSR_mon previously saved in step 2.
Finally, it exits monitor mode with a SUBS, that will make the core return in non-secure user mode, on the instruction

7. Thread 1 can resume normally.

Figure 21 illustrates in more detail the operation of the memory management logic 30 of one embodiment of the present invention. The memory management logic consists of a Memory Management Unit (MMU) 200 and a Memory Protection Unit (MPU) 220. Any memory access request issued by the core 10 that specifies a virtual address will be passed over path 234 to the MMU 200, the MMU 200 being responsible for performing predetermined access control functions, more particularly for determining the physical address corresponding to that virtual address, and for resolving access permission rights and determining region attributes.

The memory system of the data processing apparatus consists of secure memory and non-secure memory, the secure memory being used to store secure data that is intended only to be accessible by the core 10, or one or more other master devices, when that core or other device is operating in a secure mode of operation, and is accordingly operating in the secure domain.

In the embodiment of the present invention illustrated in Figure 21, the policing of attempts to access secure data in secure memory by applications running on the core 10 in non-secure mode is performed by the partition checker 222 within the MPU 220, the MPU 220 being managed by the secure operating system, also referred to herein as the secure kernel.

In accordance with preferred embodiments of the present invention a non-secure page table 58 is provided within non-secure memory, for example within a non-secure memory portion of external memory 56, and is used to store for each of a

number of non-secure memory regions defined within that page table a corresponding descriptor. The descriptor contains information from which the MMU 200 can derive access control information required to enable the MMU to perform the predetermined access control functions, and accordingly in the embodiment described with reference to Figure 21 will provide information about the virtual to physical address mapping, the access permission rights, and any region attributes.

Furthermore, in accordance with the preferred embodiments of the present invention, at least one secure page table 58 is provided within secure memory of the memory system, for example within a secure part of external memory 56, which again for a number of memory regions defined within the table provides an associated descriptor. When the processor is operating in a non-secure mode, the non-secure page table will be referenced in order to obtain relevant descriptors for use in managing memory accesses, whilst when the processor is operating in secure mode, descriptors from the secure page table will be used.

The retrieval of descriptors from the relevant page table into the MMU proceeds as follows. In the event that the memory access request issued by the core 10 specifies a virtual address, a lookup is performed in the micro-TLB 206 which stores for one of a number of virtual address portions the corresponding physical address portions obtained from the relevant page table. Hence, the micro-TLB 206 will compare a certain portion of the virtual address with the corresponding virtual address portion stored within the micro-TLB to determine if there is a match. The portion compared will typically be some predetermined number of most significant bits of the virtual address, the number of bits being dependent on the granularity of the pages within the page table 58. The lookup performed within the micro-TLB 206 will typically be relatively quick, since the micro-TLB 206 will only include a relatively few number of entries, for example eight entries

In the event that there is no match found within the micro-TLB 206, then the memory access request is passed over path 242 to the main TLB 208 which contains a number of descriptors obtained from the page tables. As will be discussed in more detail later, descriptors from both the non-secure page table and the secure page table can co-exist within the main TLB 208, and each entry within the main TLB has a

corresponding flag (referred to herein as a domain flag) which is settable to indicate whether the corresponding descriptor in that entry has been obtained from a secure page table or a non-secure page table. In any embodiments where all secure modes of operation specify physical addresses directly within their memory access requests, it will be appreciated that there will not be a need for such a flag within the main TLB, as the main TLB will only store non-secure descriptors.

Within the main TLB 208, a similar lookup process is performed to determine whether the relevant portion of the virtual address issued within the memory access request corresponds with any of the virtual address portions associated with descriptors in the main TLB 208 that are relevant to the particular mode of operation. Hence, if the core 10 is operating in non-secure mode, only those descriptors within the main TLB 208 which have been obtained from the non-secure page table will be checked, whereas if the core 10 is operating in secure mode, only the descriptors within the main TLB that have been obtained from the secure page table will be checked.

If there is a hit within the main TLB as a result of that checking process, then the access control information is extracted from the relevant descriptor and passed back over path 242. In particular, the virtual address portion and the corresponding physical address portion of the descriptor will be routed over path 242 to the micro-TLB 206, for storage in an entry of the micro-TLB, the access permission rights will be loaded into the access permission logic 202, and the region attributes will be loaded into the region attribute logic 204. The access permission logic 202 and region attribute logic 204 may be separate to the micro-TLB, or may be incorporated within the micro-TLB.

At this point, the MMU 200 is then able to process the memory access request since there will now be a hit within the micro-TLB 206. Accordingly, the micro-TLB 206 will generate the physical address, which can then be output over path 238 onto the system bus 40 for routing to the relevant memory, this being either on-chip memory such as the TCM 36, cache 38, etc, or one of the external memory units accessible via the external bus interface 42. At the same time, the access permission logic 202 will determine whether the memory access is allowed, and will issue an

abort signal back to the core 10 over path 230 if it determines that the core is not allowed to access the specified memory location in its current mode of operation. For example, certain portions of memory, whether in secure memory or non-secure memory, may be specified as only being accessible by the core when that core is operating in supervisor mode, and accordingly if the core 10 is seeking to access such a memory location when in, for example, user mode, the access permission logic 202 will detect that the core 10 does not currently have the appropriate access rights, and will issue the abort signal over path 230. This will cause the memory access to be aborted. Finally, the region attribute logic 204 will determine the region attributes for the particular memory access, such as whether the access is cacheable, bufferable, etc, and will issue such signals over path 232, where they will then be used to determine whether the data the subject of the memory access request can be cached, for example within the cache 38, whether in the event of a write access the write data can be buffered, etc.

In the event that there was no hit within the main TLB 208, then the translation table walk logic 210 is used to access the relevant page table 58 in order to retrieve the required descriptor over path 248, and then pass that descriptor over path 246 to the main TLB 208 for storage therein. The base address for both the non-secure page table and the secure page table will be stored within registers of CP15 34, and the current domain in which the processor core 10 is operating, i.e. secure domain or non-secure domain, will also be set within a register of CP15, that domain status register being set by the monitor mode when a transition occurs between the non-secure domain and the secure domain, or vice versa. The content of the domain status register will be referred to herein as the domain bit. Accordingly, if a translation table walk process needs to be performed, the translation table walk logic 210 will know in which domain the core 10 is executing, and accordingly which base address to use to access the relevant table. The virtual address is then used as an offset to the base address in order to access the appropriate entry within the appropriate page table in order to obtain the required descriptor.

Once the descriptor has been retrieved by the translation table walk logic 210, and placed within the main TLB 208, a hit will then be obtained within the main TLB, and the earlier described process will be invoked to retrieve the access control

information, and store it within the micro-TLB 206, the access permission logic 202 and the region attribute logic 204. The memory access can then be actioned by the MMU 200.

As mentioned earlier, in preferred embodiments, the main TLB 208 can store descriptors from both the secure page table and the non-secure page table, but the memory access requests are only processed by the MMU 200 once the relevant information is stored within the micro-TLB 206. In preferred embodiments, the transfer of information between the main TLB 208 and the micro-TLB 206 is monitored by the partition checker 222 located within the MPU 220 to ensure that, in the event that the core 10 is operating in a non-secure mode, no access control information is transferred into the micro-TLB 206 from descriptors in the main TLB 208 if that would cause a physical address to be generated which is within secure memory.

The memory protection unit is managed by the secure operating system, which is able to set within registers of the CP15 34 partitioning information defining the partitions between the secure memory and the non-secure memory. The partition checker 222 is then able to reference that partitioning information in order to determine whether access control information is being transferred to the micro-TLB 206 which would allow access by the core 10 in a non-secure mode to secure memory. More particularly, in preferred embodiments, when the core 10 is operating in a non-secure mode of operation, as indicated by the domain bit set by the monitor mode within the CP15 domain status register, the partition checker 222 is operable to monitor via path 244 any physical address portion seeking to be retrieved into the micro-TLB 206 from the main TLB 208 and to determine whether the physical address that would then be produced for the virtual address based on that physical address portion would be within the secure memory. In such circumstances, the partition checker 222 will issue an abort signal over path 230 to the core 10 to prevent the memory access from taking place.

It will be appreciated that in addition the partition checker 222 can be arranged to actually prevent that physical address portion from being stored in the micro-TLB 206 or alternatively the physical address portion may still be stored within the micro-

TLB 206, but part of the abort process would be to remove that incorrect physical address portion from the micro-TLB 206, for example by flushing the micro-TLB 206.

Whenever the core 10 changes via the monitor mode between a non-secure mode and a secure mode of operation, the monitor mode will change the value of the domain bit within the CP15 domain status register to indicate the domain into which the processor's operation is changing. As part of the transfer process between domains, the micro-TLB 206 will be flushed and accordingly the first memory access following a transition between secure domain and non-secure domain will produce a miss in the micro-TLB 206, and require access information to be retrieved from main TLB 208, either directly, or via retrieval of the relevant descriptor from the relevant page table.

By the above approach, it will be appreciated that the partition checker 222 will ensure that when the core is operating in the non-secure domain, an abort of a memory access will be generated if an attempt is made to retrieve into the micro-TLB 206 access control information that would allow access to secure memory.

If in any modes of operation of the processor core 10, the memory access request is arranged to specify directly a physical address, then in that mode of operation the MMU 200 will be disabled, and the physical address will pass over path 236 into the MPU 220. In a secure mode of operation, the access permission logic 224 and the region attribute logic 226 will perform the necessary access permission and region attribute analysis based on the access permission rights and region attributes identified for the corresponding regions within the partitioning information registers within the CP15 34. If the secure memory location seeking to be accessed is within a part of secure memory only accessible in a certain mode of operation, for example secure privileged mode, then an access attempt by the core in a different mode of operation, for example a secure user mode, will cause the access permission logic 224 to generate an abort over path 230 to the core in the same way that the access permission logic 202 of the MMU would have produced an abort in such circumstances. Similarly, the region attribute logic 226 will generate cacheable and bufferable signals in the same way that the region attribute logic 204 of the MMU

would have generated such signals for memory access requests specified with virtual addresses. Assuming the access is allowed, the access request will then proceed over path 240 onto the system bus 40, from where it is routed to the appropriate memory unit.

For a non-secure access where the access request specifies a physical address, the access request will be routed via path 236 into the partition checker 222, which will perform partition checking with reference to the partitioning information in the CP15 registers 34 in order to determine whether the physical address specifies a location within secure memory, in which event the abort signal will again be generated over path 230.

The above described processing of the memory management logic will now be described in more detail with reference to the flow diagrams of Figures 23 and 24. Figure 23 illustrates the situation in which the program running on the core 10 generates a virtual address, as indicated by step 300. The relevant domain bit within the CP15 domain status register 34 as set by the monitor mode will indicate whether the core is currently running in a secure domain or the non-secure domain. In the event that the core is running in the secure domain, the process branches to step 302, where a lookup is performed within the micro-TLB 206 to see if the relevant portion of the virtual address matches with one of the virtual address portions within the micro-TLB. In the event of a hit at step 302, the process branches directly to step 312, where the access permission logic 202 performs the necessary access permission analysis. At step 314, it is then determined whether there is an access permission violation, and if there is the process proceeds to step 316, where the access permission logic 202 issues an abort over path 230. Otherwise, in the absence of such an access permission violation, the process proceeds from step 314 to step 318, where the memory access proceeds. In particular the region attribute logic 204 will output the necessary cacheable and bufferable attributes over path 232, and the micro-TLB 206 will issue the physical address over path 238 as described earlier.

If at step 302 there is a miss in the micro-TLB, then a lookup process is performed within the main TLB 208 at step 304 to determine whether the required secure descriptor is present within the main TLB. If not, then a page table walk

process is executed at step 306, whereby the translation table walk logic 210 obtains the required descriptor from the secure page table, as described earlier with reference to Figure 21. The process then proceeds to step 308, or proceeds directly to step 308 from step 304 in the event that the secure descriptor was already in the main TLB 208.

At step 308, it is determined that the main TLB now contains the valid tagged secure descriptor, and accordingly the process proceeds to step 310, where the micro-TLB is loaded with the sub-section of the descriptor that contains the physical address portion. Since the core 10 is currently running in secure mode, there is no need for the partition checker 222 to perform any partition checking function.

The process then proceeds to step 312 where the remainder of the memory access proceeds as described earlier.

In the event of a non-secure memory access, the process proceeds from step 300 to step 320, where a lookup process is performed in the micro-TLB 206 to determine whether the corresponding physical address portion from a non-secure descriptor is present. If it is, then the process branches directly to step 336, where the access permission rights are checked by the access permission logic 202. It is important to note at this point that if the relevant physical address portion is within the micro-TLB, it is assumed that there is no security violation, since the partition checker 222 effectively polices the information prior to it being stored within the micro-TLB, such that if the information is within the micro-TLB, it is assumed to be the appropriate non-secure information. Once the access permission has been checked at step 336, the process proceeds to step 338, where it is determined whether there is any violation, in which event an access permission fault abort is issued at step 316. Otherwise, the process proceeds to step 318 where the remainder of the memory access is performed, as discussed earlier.

In the event that at step 320 no hit was located in the micro-TLB, the process proceeds to step 322, where a lookup process is performed in the main TLB 208 to determine whether the relevant non-secure descriptor is present. If not, a page table walk process is performed at step 324 by the translation table walk logic 210 in order to retrieve into the main TLB 208 the necessary non-secure descriptor from the non-

secure page table. The process then proceeds to step 326, or proceeds directly to step 326 from step 322 in the event that a hit within the main TLB 208 occurred at step 322. At step 326, it is determined that the main TLB now contains the valid tagged non-secure descriptor for the virtual address in question, and then at step 328 the partition checker 222 checks that the physical address that would be generated from the virtual address of the memory access request (given the physical address portion within the descriptor) will point to a location in non-secure memory. If not, i.e. if the physical address points to a location in secure memory, then at step 330 it is determined that there is a security violation, and the process proceeds to step 332 where a secure/non-secure fault abort is issued by the partition checker 222.

If however the partition checker logic 222 determines that there is no security violation, the process proceeds to step 334, where the micro-TLB is loaded with the sub-section of the relevant descriptor that contains the physical address portion, whereafter at step 336 the memory access is then processed in the earlier described manner.

The handling of memory access requests that directly issue a physical address will now be described with reference to Figure 24. As mentioned earlier, in this scenario, the MMU 200 will be deactivated, this preferably being achieved by the setting within a relevant register of the CP15 registers an MMU enable bit, this setting process being performed by the monitor mode. Hence, at step 350 the core 10 will generate a physical address which will be passed over path 236 into the MPU 220. Then, at step 352, the MPU checks permissions to verify that the memory access being requested can proceed given the current mode of operation, i.e. user, supervisor, etc. Furthermore, if the core is operating in non-secure mode, the partition checker 222 will also check at step 352 whether the physical address is within non-secure memory. Then, at step 354, it is determined whether there is a violation, i.e. whether the access permission processing has revealed a violation, or if in non-secure mode, the partition checking process has identified a violation. If either of these violations occurs, then the process proceeds to step 356 where an access permission fault abort is generated by the MPU 220. It will be appreciated that in certain embodiments there may be no distinction between the two types of abort, whereas in alternative

embodiments the abort signal could indicate whether it relates to an access permission fault or a security fault.

If no violation is detected at step 354, the process proceeds to step 358, where the memory access to the location identified by the physical address occurs.

In preferred embodiments only the monitor mode is arranged to generate physical addresses directly, and accordingly in all other cases the MMU 200 will be active and generation of the physical address from the virtual address of the memory access request will occur as described earlier.

Figure 22 illustrates an alternative embodiment of the memory management logic in a situation where all memory access requests specify a virtual address, and accordingly physical addresses are not generated directly in any of the modes of operation. In this scenario, it will be appreciated that a separate MPU 220 is not required, and instead the partition checker 222 can be incorporated within the MMU 200. This change aside, the processing proceeds in exactly the same manner as discussed earlier with reference to Figures 21 and 23.

It will be appreciated that various other options are also possible. For example, assuming memory access requests may be issued by both secure and non-secure modes specifying virtual addresses, two MMUs could be provided, one for secure access requests and one for non-secure access requests, i.e. MPU 220 in Figure 21 could be replaced by a complete MMU. In such cases, the use of flags with the main TLB of each MMU to define whether descriptors are secure or non-secure would not be needed, as one MMU would store non-secure descriptors in its main TLB, and the other MMU would store secure descriptors in its main TLB. Of course, the partition checker would still be required to check whether an access to secure memory is being attempted whilst the core is in the non-secure domain.

If, alternatively, all memory access requests directly specified physical addresses, an alternative implementation might be to use two MPUs, one for secure access requests and one for non-secure access requests. The MPU used for non-

secure access requests would have its access requests policed by a partition checker to ensure accesses to secure memory are not allowed in non-secure modes.

As a further feature which may be provided with either the Figure 21 or the Figure 22 arrangement, the partition checker 222 could be arranged to perform some partition checking in order to police the activities of the translation table walk logic 210. In particular, if the core is currently operating in the non-secure domain, then the partition checker 222 could be arranged to check, whenever the translation table walk logic 210 is seeking to access a page table, that it is accessing the non-secure page table rather than the secure page table. If a violation is detected, an abort signal would preferably be generated. Since the translation table walk logic 210 typically performs the page table lookup by combining a page table base address with certain bits of the virtual address issued by the memory access request, this partition checking may involve, for example, checking that the translation table walk logic 210 is using a base address of a non-secure page table rather than a base address of a secure page table.

Figure 25 illustrates schematically the process performed by the partition checker 222 when the core 10 is operating in a non-secure mode. It will be appreciated that in normal operation a descriptor obtained from the non-secure page table should describe a page mapped in non-secure memory only. However, in the case of software attack, the descriptor may be tampered with in order that it now describes a section that contains both non-secure and secure regions of memory. Hence, considering the example in Figure 25, the corrupted non-secure descriptor may cover a page that includes non-secure areas 370, 372, 374 and secure areas 376, 378, 380. If the virtual address issued as part of the memory access request would then correspond to a physical address in a secure memory region, for example the secure memory region 376 as illustrated in Figure 25, then the partition checker 222 is arranged to generate an abort to prevent that access taking place. Hence, even though the non-secure descriptor has been corrupted in an attempt to gain access to secure memory, the partition checker 222 prevents the access taking place. In contrast, if the physical address that would be derived using this descriptor corresponds to a non-secure memory region, for example region 374 as illustrated in Figure 25, then the access control information loaded into the micro-TLB 206 merely identifies this non-

secure region 374. Hence, accesses within that non-secure memory region 374 can occur but no accesses into any of the secure regions 376, 378 or 380 can occur. Thus, it can be seen that even though the main TLB 208 may contain descriptors from the non-secure page table that have been tampered with, the micro-TLB will only contain physical address portions that will enable access to non-secure memory regions.

As described earlier, in embodiments where both non-secure modes and secure modes may generate memory access requests specifying virtual addresses, then the memory preferably comprises both a non-secure page table within non-secure memory, and a secure page table within secure memory. When in non-secure mode, the non-secure page table will be referenced by the translation table walk logic 210, whereas when in secure mode, the secure page table will be referenced by the translation table walk logic 210. Figure 26 illustrates these two page tables. As shown in Figure 26, the non-secure memory 390, which may for example be within external memory 56 of Figure 1, includes within it a non-secure page table 395 specified in a CP15 register 34 by reference to a base address 397. Similarly, within secure memory 400, which again may be within the external memory 56 of Figure 1, a corresponding secure page table 405 is provided which is specified within a duplicate CP15 register 34 by a secure page table base address 407. Each descriptor within the non-secure page table 395 will point to a corresponding non-secure page in non-secure memory 390, whereas each descriptor within the secure page table 405 will define a corresponding secure page in the secure memory 400. In addition, as will be described in more detail later, it is possible for certain areas of memory to be shared memory regions 410, which are accessible by both non-secure modes and secure modes.

Figure 27 illustrates in more detail the lookup process performed within the main TLB 208 in accordance with preferred embodiments. As mentioned earlier, the main TLB 208 includes a domain flag 425 which identifies whether the corresponding descriptor 435 is from the secure page table or the non-secure page table. This ensures that when a lookup process is performed, only the descriptors relevant to the particular domain in which the core 10 is operating will be checked. Figure 27 illustrates an example where the core is running in the secure domain, also referred to as the secure world. As can be seen from Figure 27, when a main TLB 208 lookup is

performed, this will result in the descriptors 440 being ignored, and only the descriptors 445 being identified as candidates for the lookup process.

In accordance with preferred embodiments, an additional process ID flag 430, also referred to herein as the ASID flag, is provided to identify descriptors from process specific page tables. Accordingly, processes P1, P2 and P3 may each have corresponding page tables provided within the memory, and further may have different page tables for non-secure operation and secure operation. Further, it will be appreciated that the processes P1, P2, P3 in the secure domain may be entirely separate processes to the processes P1, P2, P3 in the non-secure domain. Accordingly, as shown in Figure 27, in addition to checking the domain when a main TLB lookup 208 is required, the ASID flag is also checked.

Accordingly, in the example in Figure 27 where in the secure domain, process P1 is executing, this lookup process identifies just the two entries 450 within the main TLB 208, and a hit or miss is then generated dependent on whether the virtual address portion within those two descriptors matches with the corresponding portion of the virtual address issued by the memory access request. If it does, then the relevant access control information is extracted and passed to the micro-TLB 206, the access permission logic 202 and the region attribute logic 204. Otherwise, a miss occurs, and the translation table walk logic 210 is used to retrieved into the main TLB 208 the required descriptor from the page table provided for secure process P1. As will be appreciated by those skilled in the art, there are many techniques for managing the content of a TLB, and accordingly when a new descriptor is retrieved for storage in the main TLB 208, and the main TLB is already full, any one of a number of known techniques may be used to determine which descriptor to evict from the main TLB to make room for the new descriptor, for example least recently used approaches, etc.

It will be appreciated that the secure kernel used in secure modes of operation may be developed entirely separately to the non-secure operating system. However, in certain cases the secure kernel and the non-secure operating system development may be closely linked, and in such situations it may be appropriate to allow secure applications to use the non-secure descriptors. Indeed, this will allow the secure applications to have direct access to non-secure data (for sharing) by knowing only

the virtual address. This of course presumes that the secure virtual mapping and the non-secure virtual mapping are exclusive for a particular ASID. In such scenarios, the tag introduced previously (i.e. the domain flag) to distinguish between secure and non-secure descriptors will not be needed. The lookup in the TLB is instead then performed with all of descriptors available.

In preferred embodiments, the choice between this configuration of the main TLB, and the earlier described configuration with separate secure and non-secure descriptors, can be set by a particular bit provided within the CP15 control registers. In preferred embodiments, this bit would only be set by the secure kernel.

In embodiments where the secure application were directly allowed to use a non-secure virtual address, it would be possible to make a non-secure stack pointer available from the secure domain. This can be done by copying a non-secure register value identifying the non-secure stack pointer into a dedicated register within the CP15 registers 34. This will then enable the non-secure application to pass parameters via the stack according to a scheme understood by the secure application.

As described earlier, the memory may be partitioned into non-secure and secure parts, and this partitioning is controlled by the secure kernel using the CP15 registers 34 dedicated to the partition checker 222. The basic partitioning approach is based on region access permissions as definable in typical MPU devices. Accordingly, the memory is divided into regions, and each region is preferably defined with its base address, size, memory attributes and access permissions. Further, when overlapping regions are programmed, the attributes of the upper region take highest priority. Additionally, in accordance with preferred embodiments of the present invention, a new region attribute is provided to define whether that corresponding region is in secure memory or in non-secure memory. This new region attribute is used by the secure kernel to define the part of the memory that is to be protected as secure memory.

At the boot stage, a first partition is performed as illustrated in Figure 28. This initial partition will determine the amount of memory 460 allocated to the non-secure world, non-secure operating system and non-secure applications. This amount

corresponds to the non-secure region defined in the partition. This information will then be used by the non-secure operating system for its memory management. The rest of the memory 462, 464, which is defined as secure, is unknown by the non-secure operating system. In order to protect integrity in the non-secure world, the non-secure memory may be programmed with access permission for secure privileged modes only. Hence, secure applications will not corrupt the non-secure ones. As can be seen from Figure 28, following this boot stage partition, memory 460 is available for use by the non-secure operating system, memory 462 is available for use by the secure kernel, and memory 464 is available for use by secure applications.

Once the boot stage partition has been performed, memory mapping of the non-secure memory 460 is handled by the non-secure operating system using the MMU 200, and accordingly a series of non-secure pages can be defined in the usual manner. This is illustrated in Figure 29.

If a secure application needs to share memory with a non-secure application, the secure kernel can change the rights of a part of the memory to transfer artificially data from one domain to the other. Hence, as illustrated in Figure 30, the secure kernel can, after checking the integrity of a non-secure page, change the rights of that page such that it becomes a secure page 466 accessible as shared memory.

When the partition of the memory is changed, the micro-TLB 206 needs to be flushed. Hence, in this scenario, when a non-secure access subsequently occurs, a miss will occur in the micro-TLB 206, and accordingly a new descriptor will be loaded from the main TLB 208. This new descriptor will subsequently be checked by the partition checker 222 of the MPU as it is attempted to retrieve it into the micro-TLB 206, and so will be consistent with the new partition of the memory.

In preferred embodiments, the cache 38 is virtual-indexed and physical-tagged. Accordingly, when an access is performed in the cache 38, a lookup will have already been performed in the micro-TLB 206 first, and accordingly access permissions, especially secure and non-secure permissions, will have been checked. Accordingly, secure data cannot be stored in the cache 38 by non-secure applications. Access to the cache 38 is under the control of the partition checking performed by the

partition checker 222, and accordingly no access to secure data can be performed in non-secure mode.

However, one problem that could occur would be for an application in the non-secure domain to be able to use the cache operations register to invalidate, clean, or flush the cache. It needs to be ensured that such operations could not affect the security of the system. For example, if the non-secure operating system were to invalidate the cache 38 without cleaning it, any secure dirty data must be written to the external memory before being replaced. Preferably, secure data is tagged in the cache, and accordingly can be dealt with differently if desired.

In preferred embodiments, if an "invalidate line by address" operation is executed by a non-secure program, the physical address is checked by the partition checker 222, and if the cache line is a secure cache line, the operation becomes a "clean and invalidate" operation, thereby ensuring that the security of the system is maintained. Further, in preferred embodiments, all "invalidate line by index" operations that are executed by a non-secure program become "clean and invalidate by index" operations. Similarly, all "invalidate all" operations executed by a non-secure program become "clean and invalidate all" operations.

Furthermore, with reference to Figure 1, any access to the TCM 36 by the DMA 32 is controlled by the micro-TLB 206. Hence, when the DMA 32 performs a lookup in the TLB to translate its virtual address into a physical one, the earlier described flags that were added in the main TLB allow the required security checking to be performed, just as if the access request had been issued by the core 10. Further, as will be discussed later, a replica partition checker is coupled to the external bus 70, preferably being located within the arbiter/decoder block 54, such that if the DMA 32 directly accesses the memory coupled to the external bus 70 via the external bus interface 42, the replica partition checker connected to that external bus checks the validity of the access. Furthermore, in certain preferred embodiments, it would be possible to add a bit to the CP15 registers 34 to define whether the DMA controller 32 can be used in the non-secure domain, this bit only being allowed to be set by the secure kernel when operating in a privileged mode.

Considering the TCM 36, if secure data is to placed within the TCM 36, this must be handled with care. As an example, a scenario could be imagined where the non-secure operating system programs the physical address range for the TCM memory 36 so that it overlaps an external secure memory part. If the mode of operation then changes to a secure mode, the secure kernel may cause data to be stored in that overlapping part, and typically the data would be stored in the TCM 36, since the TCM 36 will typically have a higher priority than the external memory. If the non-secure operating system were then to change the setting of the physical address space for the TCM 36 so that the previous secure region is now mapped in a non-secure physical area of memory, it will be appreciated that the non-secure operating system can then access the secure data, since the partition checker will see the area as non-secure and won't assert an abort. Hence, to summarise, if the TCM is configured to act as normal local RAM and not as SmartCache, it may be possible for the non-secure operating system to read secure world data if it can move the TCM base register to non-secure physical address.

To prevent this kind of scenario, a control bit is in preferred embodiments provided within the CP15 registers 34 which is only accessible in secure privilege modes of operation, and provides two possible configurations. In a first configuration, this control bit is set to "1", in which event the TCM can only be controlled by the secure privilege modes. Hence, any non-secure access attempted to the TCM control registers within the CP15 34 will cause an undefined instruction exception to be entered. Thus, in this first configuration, both secure modes and non-secure modes can use the TCM, but the TCM is controlled only by the secure privilege mode. In the second configuration, the control bit is set to "0", in which event the TCM can be controlled by the non-secure operating system. In this case, the TCM is only used by the non-secure applications. No secure data can be stored to or loaded from the TCM. Hence, when a secure access is performed, no look-up is performed within the TCM to see if the address matched the TCM address range.

By default, it is envisaged that the TCM would be used only by non-secure operating systems, as in this scenario the non-secure operating system would not need to be changed.

As mentioned earlier, in addition to the provision of the partition checker 222 within the MPU 220, preferred embodiments of the present invention also provide an analogous partition checking block coupled to the external bus 70, this additional partition checker being used to police accesses to memory by other master devices, for example the digital signal processor (DSP) 50, the DMA controller 52 coupled directly to the external bus, the DMA controller 32 connectable to the external bus via the external bus interface 42, etc. As mentioned earlier, the entire memory system can consist of several memory units, and a variety of these may exist on the external bus 70, for example the external memory 56, boot ROM 44, or indeed buffers or registers 48, 62, 66 within peripheral devices such as the screen driver 46, I/O interface 60, key storage unit 64, etc. Furthermore, different parts of the memory system may need to be defined as secure memory, for example it may be desired that the key buffer 66 within the key storage unit 64 should be treated as secure memory. If an access to such secure memory were to be attempted by a device coupled to the external bus, then it is clear that the earlier described memory management logic 30 provided within the chip containing the core 10 would not be able to police such accesses.

Figure 31 illustrates how the additional partition checker 492 coupled to the external bus, also referred to herein as the device bus, is used. The external bus would typically be arranged such that whenever memory access requests were issued onto that external bus by devices, such as devices 470, 472, those memory access requests would also include certain signals on the external bus defining the mode of operation, for example privileged, user, etc. In accordance with preferred embodiments of the present invention the memory access request also involves issuance of a domain signal onto the external bus to identify whether the device is operating in secure mode or non-secure mode. This domain signal is preferably issued at the hardware level, and in preferred embodiments a device capable of operating in secure or non-secure domains will include a predetermined pin for outputting the domain signal onto path 490 within the external bus. For the purpose of illustration, this path 490 is shown separately to the other signal paths 488 on the external bus.

This domain signal, also referred to herein as the "S bit" will identify whether the device issuing the memory access request is operating in secure domain or non-secure domain, and this information will be received by the partition checker 492

coupled to the external bus. The partition checker 492 will also have access to the partitioning information identifying which regions of memory are secure or non-secure, and accordingly can be arranged to only allow a device to have access to a secure part of memory if the S bit is asserted to identify a secure mode of operation.

By default, it is envisaged that the S bit would be unasserted, and accordingly a pre-existing non-secure device, such as device 472 illustrated in Figure 31, would not output an asserted S bit and accordingly would never be granted access by the partition checker 492 to any secure parts of memory, whether that be within registers or buffers 482, 486 within the screen driver 480, the I/O interface 484, or within the external memory 474.

For the sake of illustration, the arbiter block 476 used to arbitrate between memory access requests issued by master devices, such as devices 470, 472, is illustrated separately to the decoder 478 used to determine the appropriate memory device to service the memory access request, and separate from the partition checker 492. However, it will be appreciated that one or more of these components may be integrated within the same unit if desired.

Figure 32 illustrates an alternative embodiment, in which a partition checker 492 is not provided, and instead each memory device 474, 480, 484 is arranged to police its own memory access dependent on the value of the S bit. Accordingly, if device 470 were to assert a memory access request in non-secure mode to a register 482 within the screen driver 480 that was marked as secure memory, then the screen driver 480 would determine that the S bit was not asserted, and would not process the memory access request. Accordingly, it is envisaged that with appropriate design of the various memory devices, it may be possible to avoid the need for a partition checker 492 to be provided separately on the external bus.

Figure 33 shows different modes and applications running on a processor. The dashed lines indicate how different modes and/or applications can be separated and isolated from one another during monitoring of the processor according to an embodiment of the present invention.

The ability to monitor a processor to locate possible faults and discover why an application is not performing as expected is extremely useful and many processors provide such functions. The monitoring can be performed in a variety of ways including debug and trace functions.

In the processor according to the present technique debug can operate in several modes including halt debug mode and monitor debug mode. These modes are intrusive and cause the program running at the time to be stopped. In halt debug mode, when a breakpoint or watchpoint occurs, the core is stopped and isolated from the rest of the system and the core enters debug state. On entry the core is halted, the pipeline is flushed and no instructions are pre-fetched. The PC is frozen and any interrupts (IRQ and FIQ) are ignored. It is then possible to examine the core internal state (via the JTAG serial interface) as well as the state of the memory system. This state is invasive to program execution, as it is possible to modify current mode, change register contents, etc. Once Debug is terminated, the core exits from the Debug State by scanning in the Restart instruction through the Debug TAP (test access port). Then the program resumes execution.

In monitor debug mode, a breakpoint or watchpoint causes the core to enter abort mode, taking prefetch or Data Abort vectors respectively. In this case, the core is still in a functional mode and is not stopped as it is in Halt debug mode. The abort handler communicates with a debugger application to access processor and coprocessor state or dump memory. A debug monitor program interfaces between the debug hardware and the software debugger. If bit 11 of the debug status and control register DSCR is set (see later), interrupts (FIQ and IRQ) can be inhibited. In monitor debug mode, vector catching is disabled on Data Aborts and Prefetch Aborts to avoid the processor being forced into an unrecoverable state as a result of the aborts that are generated for the monitor debug mode. It should be noted that monitor debug mode is a type of debug mode and is not related to monitor mode of the processor which is the mode that supervises switching between secure world and non-secure world.

Debug can provide a snapshot of the state of a processor at a certain moment. It does this by noting the values in the various registers at the moment that a debug

initiation request is received. These values are recorded on a scan chain (541, 544 of Figure 41) and they are then serially output using a JTAG controller (18 or Figure 1).

An alternative way of monitoring the core is by trace. Trace is not intrusive and records subsequent states as the core continues to operate. Trace runs on an embedded trace macrocell (ETM) 22, 26 of Figure 1. The ETM has a trace port through which the trace information is exported, this is then analysed by an external trace port analyser.

The processor of embodiments of the present technique operates in two separate domains, in the embodiments described these domains comprise secure and non-secure domains. However, for the purposes of the monitoring functions, it will be clear to the skilled person that these domains can be any two domains between which data should not leak. Embodiments of the present technique are concerned with preventing leakage of data between the two domains and monitoring functions such as debug and trace which are conventionally allowed access to the whole system are a potential source of data leakage between the domains.

In the example given above of a secure and non-secure domain or world, secure data must not be available to the non-secure world. Furthermore, if debug is permitted, in secure world, it may be advantageous for some of the data within secure world to be restricted or hidden. The hashed lines in Figure 33 shows some examples of possible ways to segment data access and provide different levels of granularity. In Figure 33, monitor mode is shown by block 500 and is the most secure of all the modes and controls switching between secure and non-secure worlds. Below monitor mode 500 there is a supervisor mode, this comprises secure supervisor mode 510 and non-secure supervisor mode 520. Then there is non-secure user mode having applications 522 and 524 and secure user mode with applications 512, 514 and 516. The monitoring modes (debug and trace) can be controlled to only monitor non-secure mode (to the left of hashed line 501). Alternatively the non-secure domain or world and the secure user mode may be allowed to be monitored (left of 501 and the portion right of 501 that lies below 502). In a further embodiment the non-secure world and certain applications running in the secure user domain may be allowed, in this case further segmentation by hashed lines 503 occurs. Such divisions help prevent leakage

of secure data between different users who may be running the different applications. In some controlled cases monitoring of the entire system may be allowed. According to the granularity required the following parts of the core need to have their access controlled during monitoring functions.

There are four registers that can be set on a Debug event; the instruction Fault Status Register (IFSR), Data Fault Status Register (DFSR), Fault Address Register (FAR), and Instruction Fault Address Register (IFAR). These registers should be flushed in some embodiments when going from secure world to non-secure world to avoid any leak of data.

PC sample register: The Debug TAP can access the PC through scan chain 7. When debugging in secure world, that value may be masked depending on the debug granularity chosen in secure world. It is important that non-secure world, or non-secure world plus secure user applications cannot get any value of the PC while the core is running in the secure world.

TLB entries: Using CP15 it is possible to read micro TLB entries and read and write main TLB entries. We can also control main TLB and micro TLB loading and matching. This kind of operation must be strictly controlled, particularly if secure thread-aware debug requires assistance of the MMU/MPU.

Performance Monitor Control register: The performance control register gives information on the cache misses, micro TLB misses, external memory requests, branch instruction executed, etc. Non-secure world should not have access to this data, even in Debug State. The counters should be operable in secure world even if debug is disabled in secure world.

Debugging in cache system: Debugging must be non-intrusive in a cached system. It is important is to keep coherency between cache and external memory. The Cache can be invalidated using CP15, or the cache can be forced to be write-through in all regions. In any case, allowing the modification of cache behaviour in debug can be a security weakness and should be controlled.

Endianness: Non-secure world or secure user applications that can access to debug should not be allowed to change endianness. Changing the endianness could cause the secure kernel to malfunction. Endianness access is prohibited in debug, according to the granularity.

Access of the monitoring functions to portions of the core can be controlled at initiation of the monitoring function. Debug and trace are initialised in a variety of ways. Embodiments of the present technique control the access of the monitoring function to certain secure portions of the core by only allowing initialisation under certain conditions.

Embodiments of the present technique seek to restrict entry into monitoring functions with the following granularity:

By controlling seperately intrusive and observable (trace) debug;

By allowing debug entry in secure user mode only or in the whole secure world;

By allowing debug in secure user mode only and moreover taking account of the thread ID (application running).

In order to control the initiation of a monitoring function it is important to be aware of how the functions can be initiated. Figure 34 shows a table illustrating the possible ways of initiating a monitoring function, the type of monitoring function that is initiated and the way that such an initiation instruction can be programmed.

Generally, these monitoring instructions can be entered via software or via hardware, i.e. via the JTAG controller. In order to control the initiation of monitoring functions, control values are used. These comprise enable bits which are condition dependent and thus, if a particular condition is present, monitoring is only allowed to start if the enable bit is set. These bits are stored on a secure register CP14 (debug and status control register, DSCR), which is located in ICE 530 (see Figure 41).

In a preferred embodiment there are four bits that enable/disable intrusive and observable debug, these comprise a secure debug enable bit, a secure trace enable bit, a secure user-mode enable bit and a secure thread aware enable bit. These control

values serve to provide a degree of controllable granularity for the monitoring function and as such can help stop leakage of data from a particular domain. Figure 35 provides a summary of these bits and how they can be accessed.

These control bits are stored in a register in the secure domain and access to this register is limited to three possibilities. Software access is provided via ARM coprocessor MRC/MCR instructions and these are only allowed from the secure supervisor mode. Alternatively, software access can be provided from any other mode with the use of an authentication code. A further alternative relates more to hardware access and involves the instructions being written via an input port on the JTAG. In addition to being used to input control values relating to the availability of monitoring functions, this input port can be used to input control values relating to other functions of the processor.

Further details relating to the scan chain and JTAG are given below.

Register logic cell

Every integrated circuit (IC) consists of two kind of logic:

- Combinatory logic cells; like AND, OR, INV gates. Such gates or combination of such gates is used to calculate Boolean expressions according to one or several input signals.
- Register logic cells; like LATCH, FLIP-FLOP. Such cells are used to memorize any signal value. Figure 36 shows a positive-edge triggered FLIP-FLOP view:

When positive-edge event occurs on the clock signal (CK), the output (Q) received the value of the input (D); otherwise the output (Q) keeps its value in memory.

Scan chain cell

For test or debug purpose, it is required to bypass functional access of register logic cells and to have access directly to the contents of the register logic cells. Thus register cells are integrated in a scan chain cell as shown in Figure 37.

In functional mode, SE (Scan Enable) is clear and the register cell works as a single register cell. In test or debug mode, SE is set and input data can come from SI input (Scan In) instead of D input.

Scan chain

All scan chain cells are chained in scan chain as shown in figure 38.

In functional mode, SE is clear and all register cells can be accessed normally and interact with other logic of the circuit. In Test or Debug mode, SE is set and all registers are chained between each other in a scan chain. Data can come from the first scan chain cell and can be shifted through any other scan chain cell, at the cadence of each clock cycle. Data can be shifted out also to see the contents of the registers.

TAP controller

A debug TAP controller is used to handle several scan chains. The TAP controller can select a particular scan chain: it connects "Scan In" and "Scan Out" signals to that particular scan-chain. Then data can be scanned into the chain, shifted, or scanned out. The TAP controller is controlled externally by a JTAG port interface. Figure 39 schematically illustrates a TAP controller

JTAG Selective Disable Scan Chain Cell

For security reasons, some registers might not be accessible by scan chain, even in debug or test mode. A new input called JADI (JTAG Access Disable) can allow removal dynamically or statically of a scan chain cell from a whole scan chain, without modifying the scan chain structure in the integrated circuit. Figures 40A and B schematically show this input.

If JADI is inactive (JADI = 0), whether in functional or test or debug mode, the scan chain works as usual. If JADI is active (JADI = 1), and if we are in test or debug mode, some scan chain cells (chosen by designer), may be "removed" from the scan chain structure. In order to keep the same number of scan-chain cell, the JTAG Selective Disable Scan Chain Cell use a bypass register. Note that Scan Out (SO) and scan chain cell output (Q) are now different.

Figure 41 schematically shows the processor including parts of the JTAG. In normal operation instruction memory 550 communicates with the core and can under certain circumstances also communicate with register CP14 and reset the control values. This is generally only allowable from secure supervisor mode.

When debug is initiated instructions are input via debug TAP 580 and it is these that control the core. The core in debug runs in a step by step mode. Debug TAP has access to CP14 via the core (in dependence upon an access control signal input on the JSDAEN pin shown as JADI pin, JTAG ACCESS DISABLE INPUT in Figure 40) and the control values can also be reset in this way.

Access to the CP14 register via debug TAP 580 is controlled by an access control signal JSDAEN. This is arranged so that in order for access and in particular write access to be allowed JSDAEN must be set high. During board stage when the whole processor is being verified, JSDAEN is set high and debug is enabled on the whole system. Once the system has been checked, the JSDAEN pin can be tied to ground, this means that access to the control values that enable debug in secure mode is now not available via Debug TAP 580. Generally processors in production mode have JSDAEN tied to ground. Access to the control values is thus, only available via the software route via instruction memory 550. Access via this route is limited to secure supervisor mode or to another mode provided an authentication code is given (see Figure 42).

It should be noted that by default debug (intrusive and observable – trace) are only available in non-secure world. To enable them to be available in secure world the control value enable bits need to be set.

The advantages of this are that debug can always be initiated by users to run in non-secure world. Thus, although access to secure world is not generally available to users in debug this may not be a problem in many cases because access to this world is limited and secure world has been fully verified at board stage prior to being made available. It is therefore foreseen that in many cases debugging of the secure world will not be necessary. A secure supervisor can still initiate debug via the software route of writing CP14 if necessary.

Figure 42 schematically shows the control of debug initialisation. In this figure a portion of the core 600 comprises a storage element 601 (which may be a CP15 register as previously discussed) in which is stored a secure status bit S indicative of whether the system is in secure world or not. Core 600 also comprises a register 602 comprising bits indicative of the mode that the processor is running in, for example user mode, and a register 603 providing a context identifier that identifies the application or thread that is currently running on the core.

When a breakpoint is reached comparator 610, which compares a breakpoint stored on register 611with the address of the core stored in register 612, sends a signal to control logic 620. Control logic 620 looks at the secure state S, the mode 602 and the thread (context identifier) 603 and compares it with the control values and condition indicators stored on register CP14. If the system is not operating in secure world, then a "enter debug" signal will be output at 630. If however, the system is operating in secure world, the control logic 620 will look at the mode 602, and if it is in user mode will check to see if user mode enable and debug enable bits are set. If they are then debug will be initialised provided that a thread aware bit has not been initialised. The above illustrates the hierarchical nature of the control values.

The thread aware portion of the monitoring control is also shown schematically in Figure 42 along with how the control value stored in register CP14 can only be changed from secure supervisor mode (in this embodiment the processor is in production stage and JSDAEN is tied to ground). From a secure user mode, secure supervisor mode can be entered using an authentication code and then the control value can be set in CP14.

Control logic 620 outputs an "enter debug" signal when address comparator 610 indicates that a breakpoint has been reached provided thread comparator 640 shows that debug is allowable for that thread. This assumes that the thread aware initialisation bit is set in CP14. If the thread aware initialisation bit is set following a breakpoint, debug or trace can only be entered if address and context identifiers match those indicated in the breakpoint and in the allowable thread indicator. Following initiation of a monitoring function, the capture of diagnostic data will only continue while the context identifier is detected by comparator 640 as an allowed thread. When a context identifier shows that the application running is not an allowed one, then the capture of diagnostic data is suppressed.

It should be noted that in the preferred embodiment, there is some hierarchy within the granularity. In effect the secure debug or trace enable bit is at the top, followed by the secure user-mode enable bit and lastly comes the secure thread aware enable bit. This is illustrated in Figures 43A and 43B (see below).

The control values held in the "Debug and Status Control" register (CP14) control secure debug granularity according to the domain, the mode and the executing thread. It is on top of secure supervisor mode. Once the "Debug and Status Control" register CP14 is configured, it's up to secure supervisor mode to program the corresponding breakpoints, watchpoints, etc to make the core enter Debug State.

Figure 43A shows a summary of the secure debug granularity for intrusive debug. Default values at reset are represented in grey colour.

It is the same for debug granularity concerning observable debug. Figure 43B shows a summary of secure debug granularity in this case, here default values at reset are also represented in grey colour.

Note that Secure user-mode debug enable bit and Secure thread-aware debug enable bit are commonly used for intrusive and observable debug.

A thread aware initialisation bit is stored in register CP14 and indicates if granularity by application is required. If the thread aware bit has been initialised, the

control logic will further check that the application identifier or thread 603 is one indicated in the thread aware control value, if it is, then debug will be initialised. If either of the user mode or debug enable bits are not set or the thread aware bit is set and the application running is not one indicated in the thread aware control value, then the breakpoint will be ignored and the core will continue doing what it was doing and debug will not be initialised.

In addition to controlling initialisation of monitoring functions, the capture of diagnostic data during a monitor function can also be controlled in a similar way. In order to do this the core must continue to consider both the control values, i.e. the enable bits stored in register CP14 and the conditions to which they relate during operation of the monitoring function.

Figure 44 shows schematically granularity of a monitoring function while it is running. In this case region A relates to a region in which it is permissible to capture diagnostic data and region B relates to region in which control values stored in CP14 indicate that it is not possible to capture diagnostic data.

Thus, when debug is running and a program is operating in region A, diagnostic data is output in a step-by-step fashion during debug. When operation switches to Region B, where the capture of diagnostic data is not allowed, debug no longer proceeds in a step by step fashion, rather it proceeds atomically and no data is captured. This continues until operation of the program re-enters region A whereupon the capture of diagnostic data starts again and debug continues running in a step-by-step fashion.

In the above embodiment, if secure domain is not enabled, a SMI instruction is always seen as an atomic event and the capture of diagnostic data is suppressed.

Furthermore, if the thread aware initialisation bit is set then granularity of the monitoring function during operation with respect to application also occurs.

With regard to observable debug or trace, this is done by ETM and is entirely independent of debug. When trace is enabled ETM works as usual and when it is disabled, ETM hides trace in the secure world, or part of the secure world depending on the granularity chosen. One way to avoid ETM capturing and tracing diagnostic data in the secure domain when this is not enabled is to stall ETM when the S-bit is high. This can be done by combining the S-bit with the ETMPWRDOWN signal, so that the ETM values are held at their last values when the core enters secure world. The ETM should thus trace a SMI instruction and then be stalled until the core returns to non-secure world. Thus, the ETM would only see non-secure activity.

A summary of some of the different monitoring functions and their granularity is given below.

Intrusive debug at board stage

At board stage when the JSDAEN pin is not tied, there is the ability to enable debug everywhere before starting any boot session. Similarly, if we are in secure supervisor mode we have similar rights.

If we initialise debug in halt debug mode all registers are accessible (non-secure and secure register banks) and the whole memory can be dumped, except the bits dedicated to control debug.

Debug halt mode can be entered from whatever mode and from whatever domain. Breakpoints and watchpoints can be set in secure or in non-secure memory. In debug state, it is possible to enter secure world by simply changing the S bit via an MCR instruction.

As debug mode can be entered when secure exceptions occur, the vector trap register is extended with new bits which are;

SMI vector trapping enable

Secure data abort vector trapping enable

Secure prefetch abort vector trapping enable

Secure undefined vector trapping enable.

In monitor debug mode, if we allow debug everywhere, even when an SMI is called in non-secure world, it is possible to enter secure world in step-by-step debug. When a breakpoint occurs in secure domain, the secure abort handler is operable to dump secure register bank and secure memory.

The two abort handlers in secure and in non-secure world give their information to the debugger application so that debugger window (on the associated debug controlling PC) can show the register state in both secure and non-secure worlds.

Figure 45A shows what happens when the core is configured in monitor debug mode and debug is enabled in secure world. Figure 45B shows what happens when the core is configured in monitor debug mode and the debug is disabled in secure world. This later process will be described below.

Intrusive debug at production stage

In production stage when JSDAEN is tied and debug is restricted to non-secure world, unless the secure supervisor determines otherwise, then the table shown in Figure 45B shows what happens. In this case SMI should always be considered as an atomic instruction, so that secure functions are always finished before entering debug state.

Entering debug halt mode is subject to the following restrictions:

External debug request or internal debug request is taken into account in non-secure world only. If EDBGRQ (external debug request) is asserted while in secure world, the core enters debug halt mode once secure function is terminated and the core is returned in non-secure world.

Programming a breakpoint or watchpoint on secure memory has no effect and the core is not stopped when the programmed address matches.

Vector Trap Register (details of this are given below) concerns non-secure exceptions only. All extended trapping enable bits explained before have no effect.

Once in halt debug mode the following restrictions apply:

S bit cannot be changed to force secure world entry, unless secure debug is enabled.

Mode bits can not be changed if debug is permitted in secure supervisor mode only.

Dedicated bits that control secure debug cannot be changed.

If a SMI is loaded and executed (with system speed access), the core re-enters debug state only when secure function is completely executed.

In monitor debug mode because monitoring cannot occur in secure world, the secure abort handler does not need to support a debug monitor programme. In non secure world, step-by-step is possible but whenever an SMI is executed secure function is executed entirely in other words an XWSI only "step-over" is allowed while "step-in" and "step-over" are possible on all other instructions. XWSI is thus considered an atomic instruction.

Once secure debug is disabled, we have the following restrictions:

Before entering monitor mode:

Breakpoints and watchpoints are only taken into account in non-secure world. If bit S is set, breakpoints/watchpoints are bypassed. Note that watchpoints units are also accessible with MCR/MRC (CP14) which is not a security issue as breakpoint/watchpoint has no effect in secure memory.

BKPT are normally used to replace the instruction on which breakpoint is set. This supposes to overwrite this instruction in memory by BKPT instruction, which will be possible only in non-secure mode.

Vector Trap Register concerns non-secure exceptions only. All extended trapping enable bits explained before have no effect. Data abort and Pre-fetch abort enable bits should be disabled to avoid the processor being forced in to an unrecoverable state.

Via JTAG, we have the same restrictions as for halt mode (S bit cannot be modified, etc)

Once in monitor mode (non-secure abort mode)

The non-secure abort handler can dump non-secure world and has no visibility on secure banked registers as well as secure memory.

Executes secure functions with atomic SMI instruction

S bit cannot be changed to force secure world entry.

Mode bits can not be changed if debug is permitted in secure supervisor mode only.

Note that if an external debug request (EDBGRQ) occurs,

In non-secure world, the core terminates the current instruction and enters then immediately debug state (in halt mode).

In secure world, the core terminates the current function and enters the Debug State when it has returned in non-secure world.

The new debug requirements imply some modifications in core hardware. The S bit must be carefully controlled, and the secure bit must not be inserted in a scan chain for security reason.

In summary, in debug, mode bits can be altered only if debug is enabled in secure supervisor mode. It will prevent anybody that has access to debug in the secure domain to have access to all secure world by altering the system (modifying TBL entries, etc). In that way each thread can debug its own code, and only its own code. The secure kernel must be kept safe. Thus when entering debug while the core is running in non-secure world, mode bits can only be altered as before.

Embodiments of the technique use a new **vector trap register**. If one of the bits in this register is set high and the corresponding vector triggers, the processor enters debug state as if a breakpoint has been set on an instruction fetch from the relevant exception vector. The behaviour of these bits may be different according to the value of 'Debug in Secure world Enable' bit in debug control register.

The new vector trap register comprises the following bits: D_s_abort, P_s_abort, S_undef, SMI, FIQ, IRQ, Unaligned, D_abort, P_abort, SWI and Undef.

- D_s_abort bit: should only be set when debug is enabled in secure world and when debug is configured in halt debug mode. In monitor debug mode, this bit should never bit set. If debug in secure world is disabled, this bit has no effect whatever its value.
- P_s_abort bit: same as D_s_abort bit.
- S_undef bit: should only be set when debug is enable in secure world. If debug in secure world is disabled, this bit has no effect whatever its value is.
- SMI bit: should only be set when debug is enabled in secure world. If debug in secure world is disabled, this bit has no effect whatever its value is.
- FIQ, IRQ, D_abort, P_abort, SWI, undef bits: correspond to non-secure exceptions, so they are valid even if debug in secure world is disabled. Note that D_abort and P_abort should not be asserted high in monitor mode.
- Reset bit: as we enter secure world when reset occurs, this bit is valid only when debug in secure world is enabled, otherwise it has no effect.

Although a particular embodiment of the invention has been described herein, it will be apparent that the invention is not limited thereto, and that many modifications and additions may be made within the scope of the invention. For example, various combinations of the features of the following dependent could be made with the features of the independent claims without departing from the scope of the present invention.

CLAIMS

1. Apparatus for processing data, said apparatus comprising:

a processor operable in a plurality of modes and either a secure domain or a non-secure domain including:

at least one secure mode being a mode in said secure domain; and

at least one non-secure mode being a mode in said non-secure domain; wherein

when said processor is executing a program in a secure mode said program has access to secure data which is not accessible when said processor is operating in a non-secure mode;

said processor is responsive to one or more exception conditions for triggering exception processing; and

said processor being responsive to one or more parameters specifying which of said exceptions should be handled by a secure mode exception handler executing in a secure mode and which of said exceptions should be handled by an exception handler executing in a mode within a current one of said secure domain and said non-secure domain when that exception occurs.

2. Apparatus as claimed in claim 1, wherein at least one of said parameters is stored in an exception trap mask register.

3. Apparatus as claimed in claim 2, wherein said exception trap mask register is writable when said processor is in a secure mode and said exception trap mask register is non-writable when said processor is in a non-secure mode.

4. Apparatus as claimed in any one of claims 2 and 3, further comprising a configuration controlling coprocessor associated with said processor and wherein said exception trap mask register is a register within said configuration controlling coprocessor.

5. Apparatus as claimed in any one of the preceding claims, wherein at least one of said parameters is a signal value provided at a hardware input to said processor.

6. Apparatus as claimed in any one of the preceding claims, wherein said secure exception handler is part of a secure operating system operable in said secure mode.

7. Apparatus as claimed in any one of the preceding claims, wherein said non-secure exception handler is part of a non-secure operating system operable in said non-secure mode.

8. Apparatus as claimed in any one of the preceding claims, wherein said processor is also operable in a monitor mode and any switching between a secure mode and a non-secure mode required for handling of an exception as specified by said parameters takes place via said monitor mode, said processor being operable at least partially in said monitor mode to execute a monitor program to manage switching between said secure mode and said non-secure mode.

9. Apparatus as claimed in claim 8, wherein said monitor program may change said parameters to determine where an exception should be handled.

10. Apparatus as claimed in any one of claims 8 and 9, wherein said processor includes a register bank and said monitor program is operable to flush at least a portion of said register bank shared between said secure mode and said non-secure mode when switching from said secure mode to said non-secure mode such that no secure data held within said register bank may pass from said secure mode to said non-secure mode other than as permitted by said monitor program.

11. A method of processing data, said method comprising the steps of:

executing a program with a processor operable in a plurality of modes and either a secure domain or a non-secure domain including:

at least one secure mode being a mode in said secure domain; and

at least one non-secure mode being a mode in said non-secure domain; wherein

when said processor is executing a program in a secure mode said program has access to secure data which is not accessible when said processor is operating in a non-secure mode;

in response to one or more exception conditions triggering exception processing using an exception handler; wherein

said processor selects an exception handler in response to one or more parameters specifying which of said exceptions should be handled by a secure mode exception handler executing in a secure mode and which of said exceptions should be handled by an exception handler executing in a mode within a current one of said secure domain and said non-secure domain when that exception occurs.

12. A method as claimed in claim11, wherein at least one of said parameters is stored in an exception trap mask register.

13. A method as claimed in claim 12, wherein said exception trap mask register is writable when said processor is in a secure mode and said exception trap mask register is non-writable when said processor is in a non-secure mode.

14. A method as claimed in any one of claims 12 and 13, further comprising a configuration controlling coprocessor associated with said processor and wherein said exception trap mask register is a register within said configuration controlling coprocessor.

15. A method as claimed in any one of claims 11 to 14, wherein at least one of said parameters is a signal value provided at a hardware input to said processor.

16. A method as claimed in any one of claims 11 to 15, wherein said secure exception handler is part of a secure operating system operable in said secure mode.

17. A method as claimed in any one of claims 11 to 16, wherein said non-secure exception handler is part of a non-secure operating system operable in said non-secure mode.

18. A method as claimed in any one of claims 11 to 17, wherein said processor is also operable in a monitor mode and any switching between a secure mode and a non-secure mode required for handling of an exception as specified by said parameters takes place via said monitor mode, said processor being operable at least partially in said monitor mode to execute a monitor program to manage switching between said secure mode and said non-secure mode.

19. A method as claimed in claim 18, wherein said monitor program may change said parameters to determine where an exception should be handled.

20. A method as claimed in any one of claims 18 and 19, wherein said processor includes a register bank and said monitor program is operable to flush at least a portion of said register bank shared between said secure mode and said non-secure mode when switching from said secure mode to said non-secure mode such that no secure data held within said register bank may pass from said secure mode to said non-secure mode other than as permitted by said monitor program.

ABSTRACT

EXCEPTION HANDLING CONTROL IN A SECURE PROCESSING SYSTEM

There is a provided a data processing system comprising:

a processor operable in a plurality of modes and either a secure domain or a non-secure domain including:

at least one secure mode being a mode in said secure domain; and

at least one non-secure mode being a mode in said non-secure domain;

wherein

when said processor is executing a program in a secure mode said program has access to secure data which is not accessible when said processor is operating in a non-secure mode;

said processor is responsive to one or more exception conditions for triggering exception processing; and

said processor being responsive to one or more parameters specifying which of said exceptions should be handled by a secure mode exception handler executing in a secure mode and which of said exceptions should be handled by an exception handler executing in a mode within a current one of said secure domain and said non-secure domain when that exception occurs.

[Figure 18]

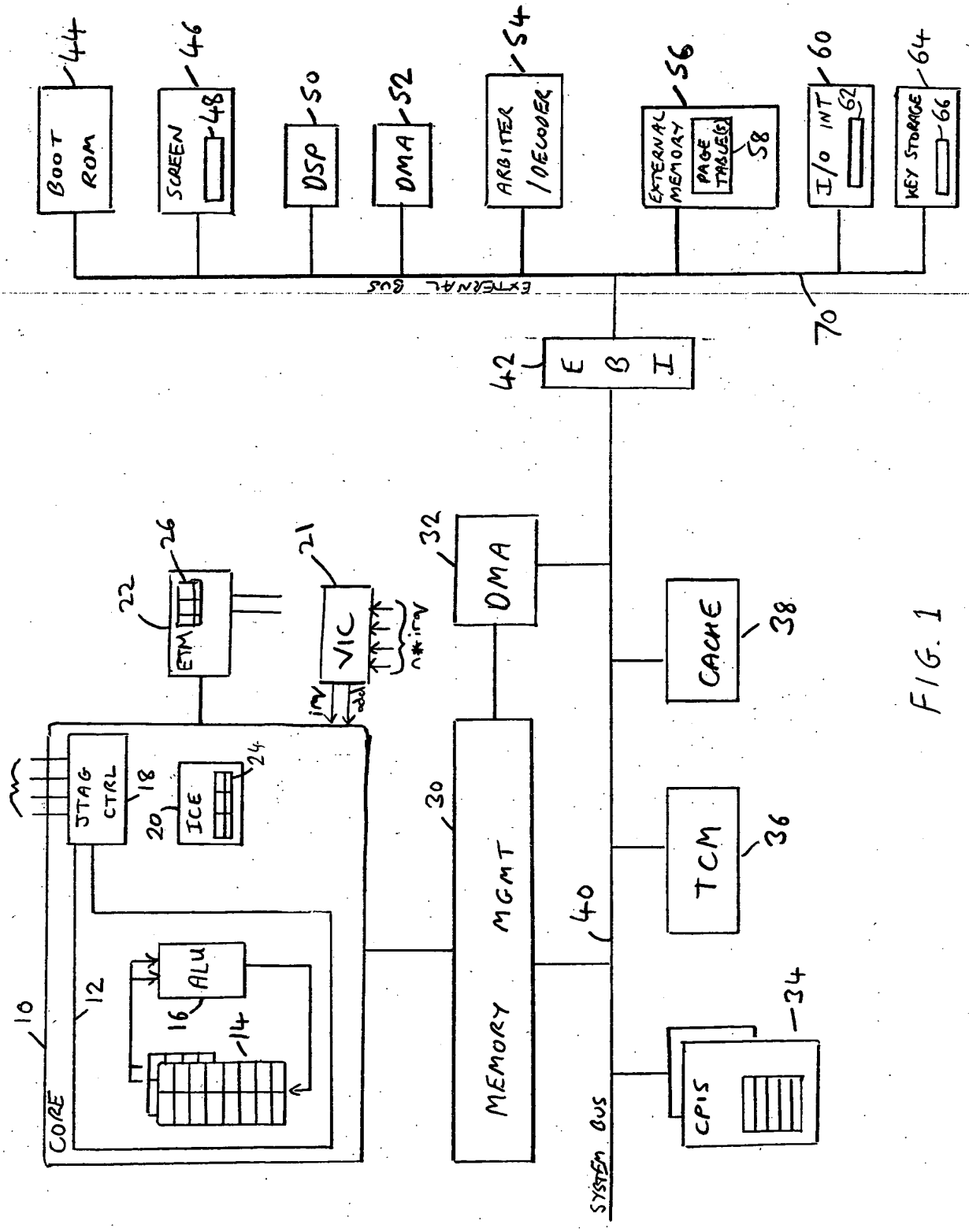


FIG. 1

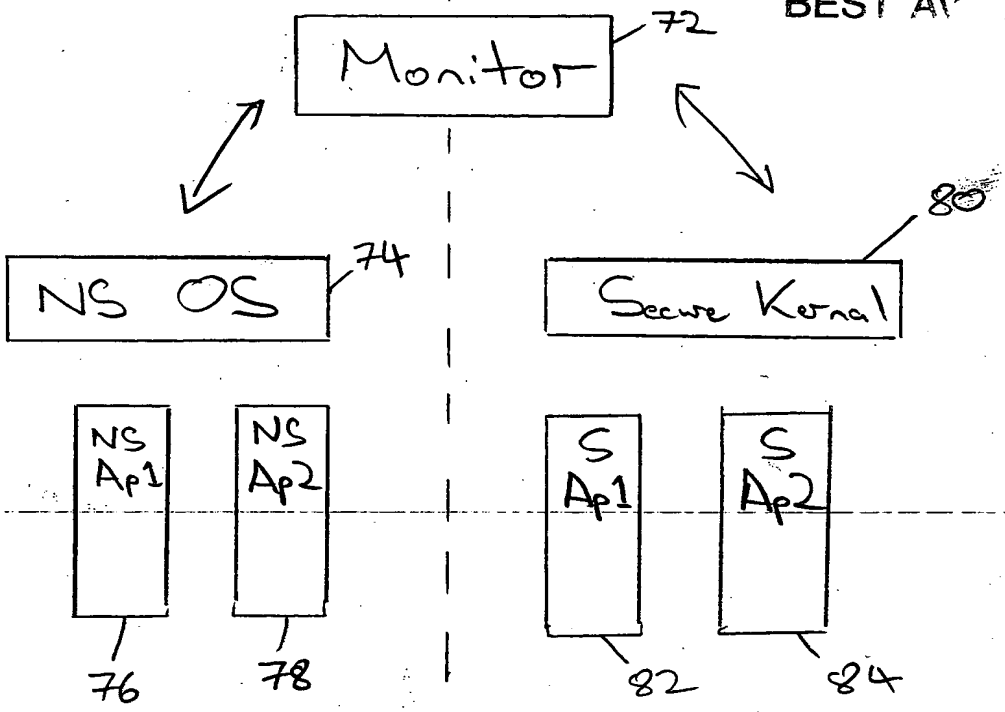


Fig. 2

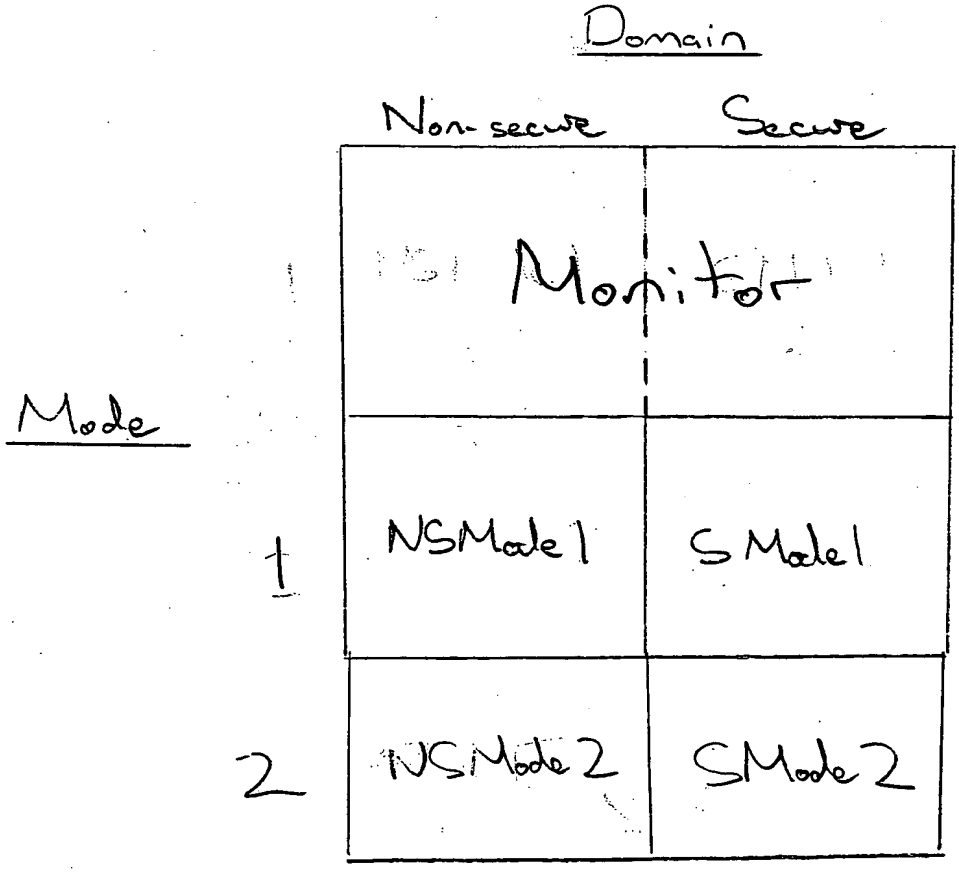


Fig. 3

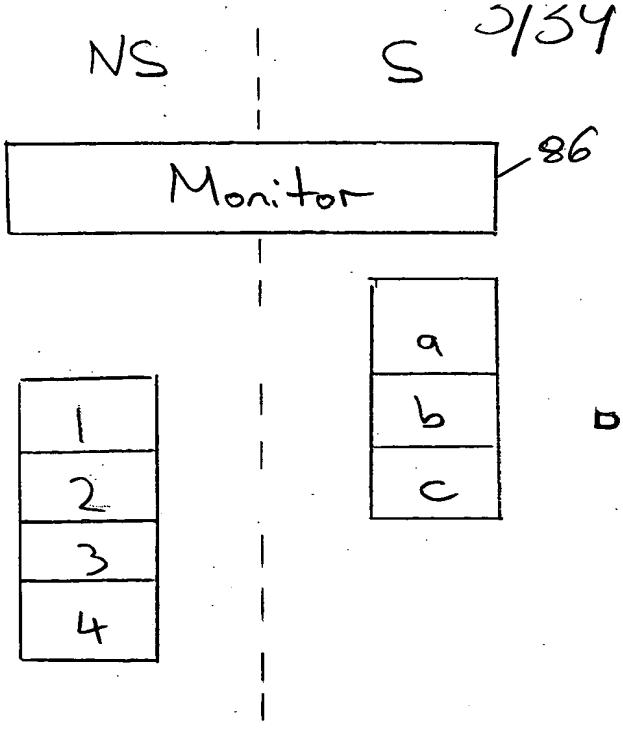


Fig. 4

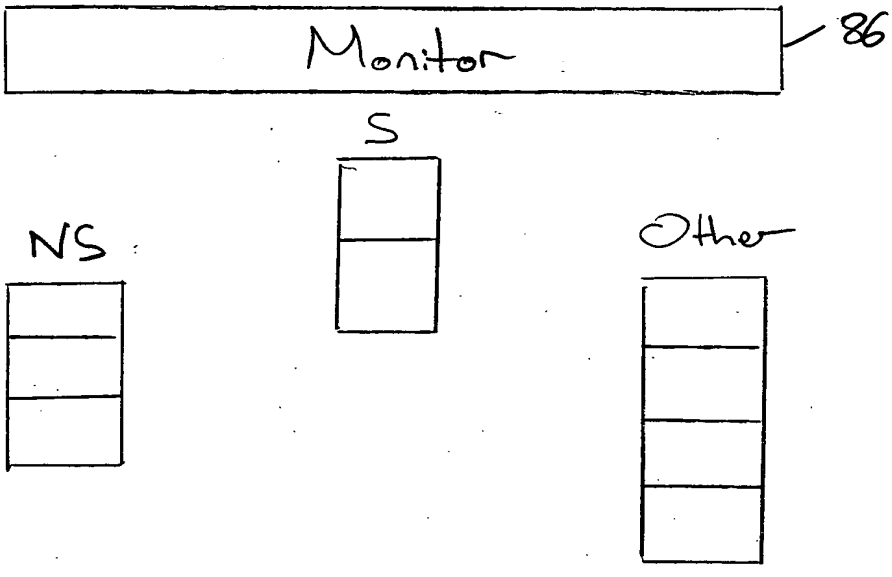
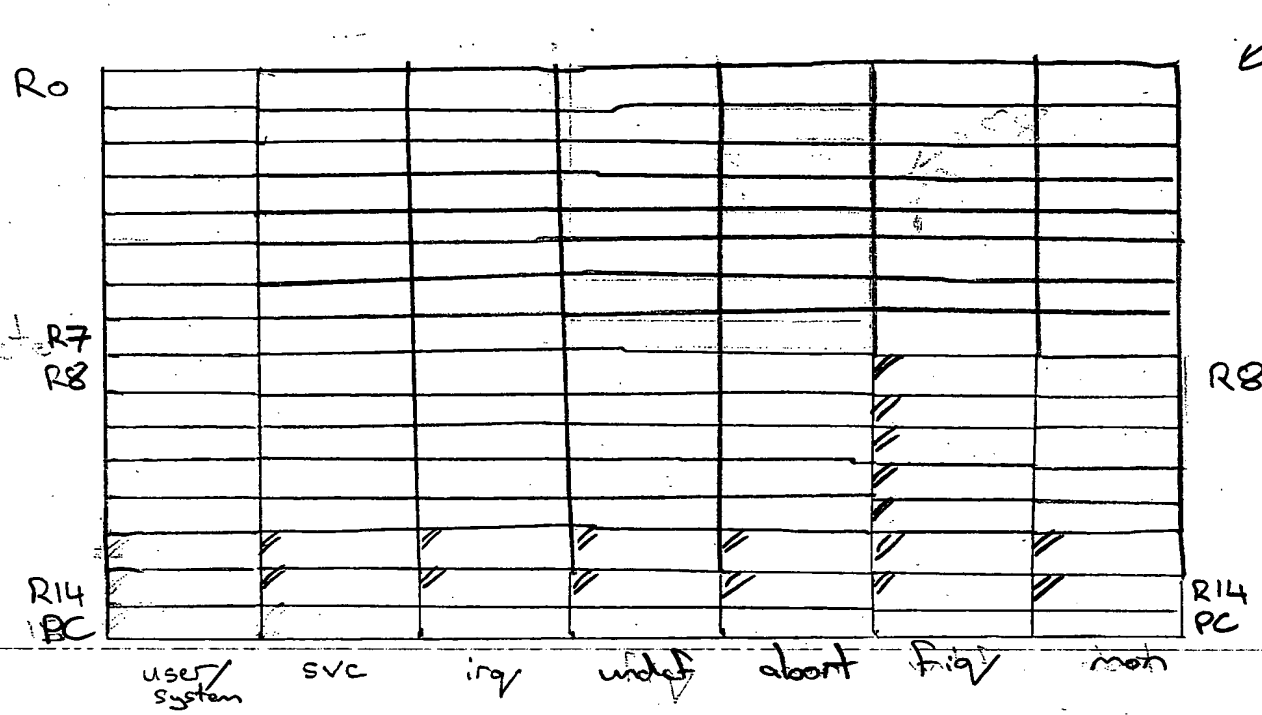


Fig. 5



CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	SPSR_svc	SPSR_irq	SPSR_und	SPSR_abt	SPSR_fiq	SPSR_mon

// = private to mode

Fig. 6

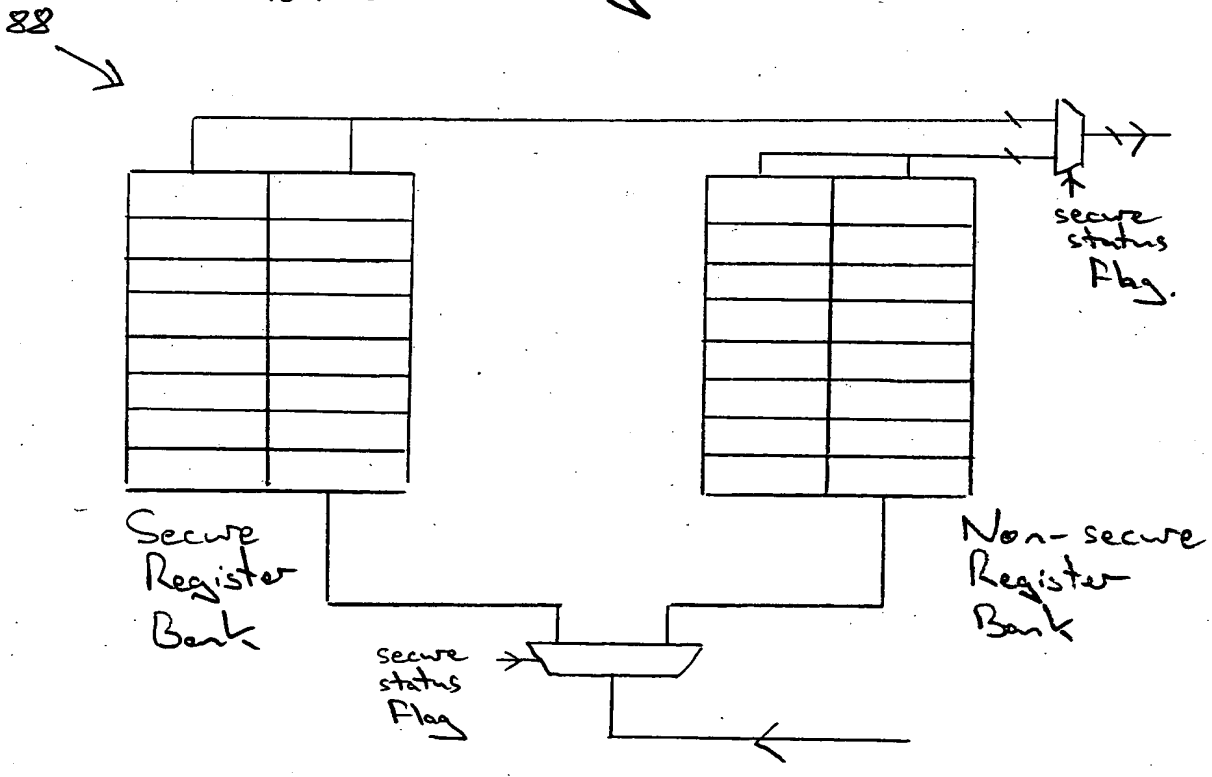


Fig. 7

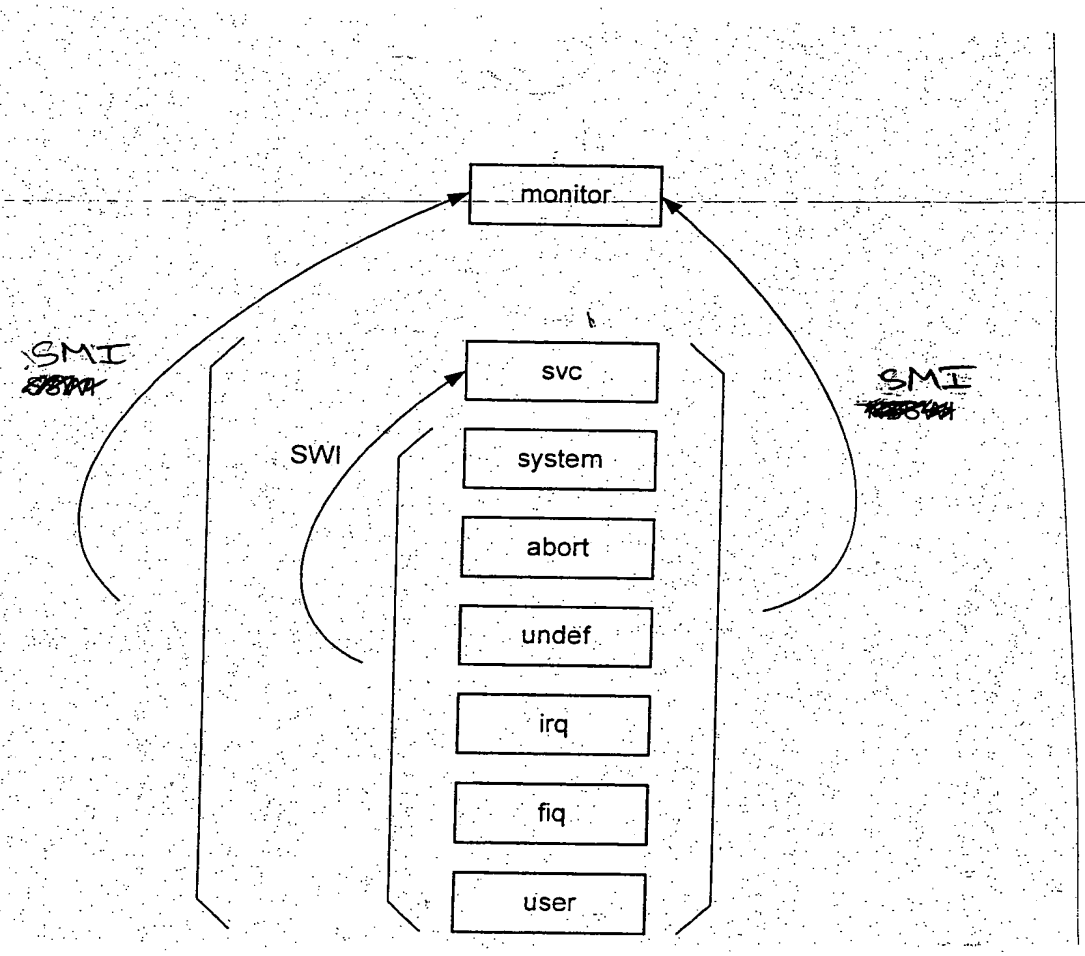


Fig. 8

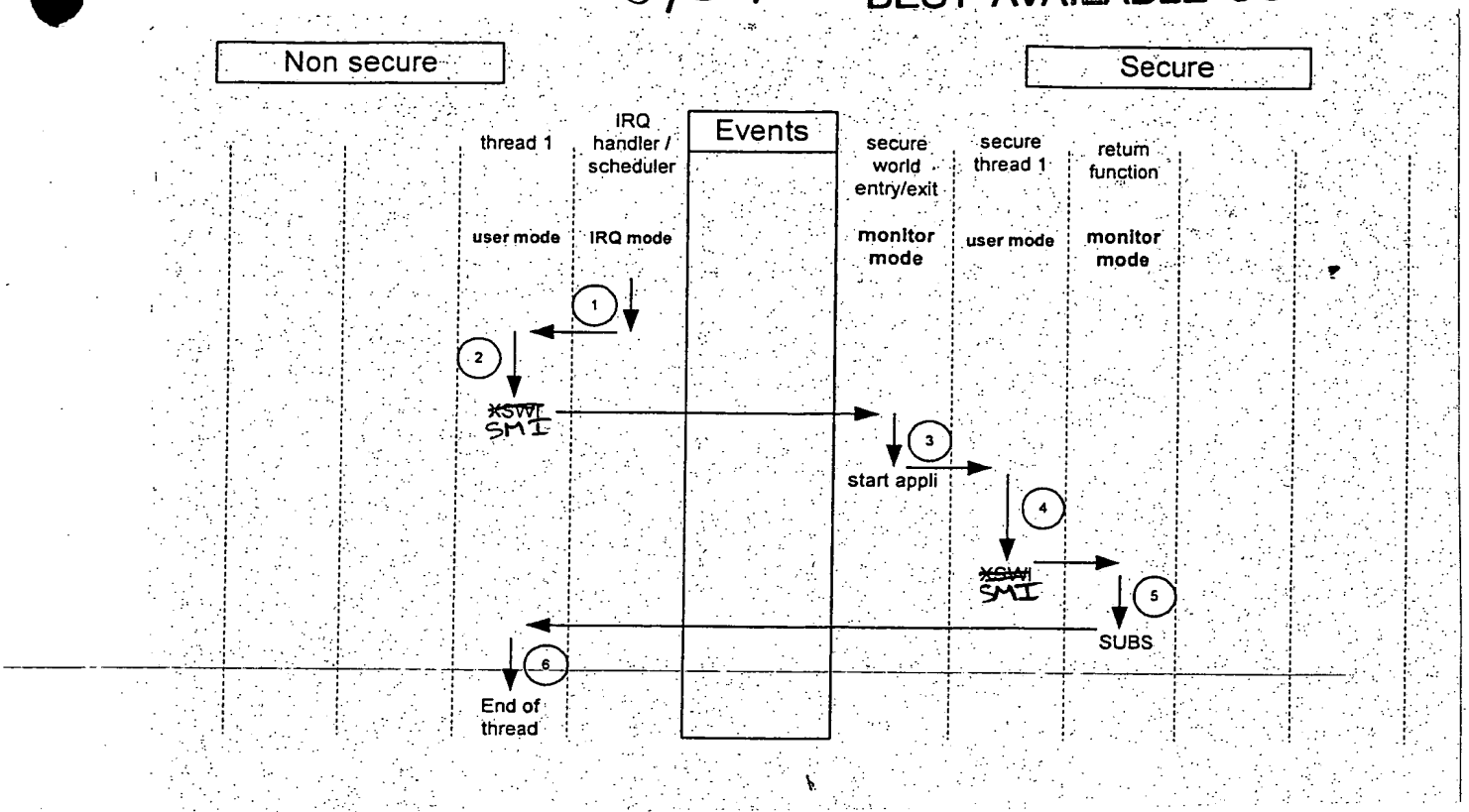


Fig. 9

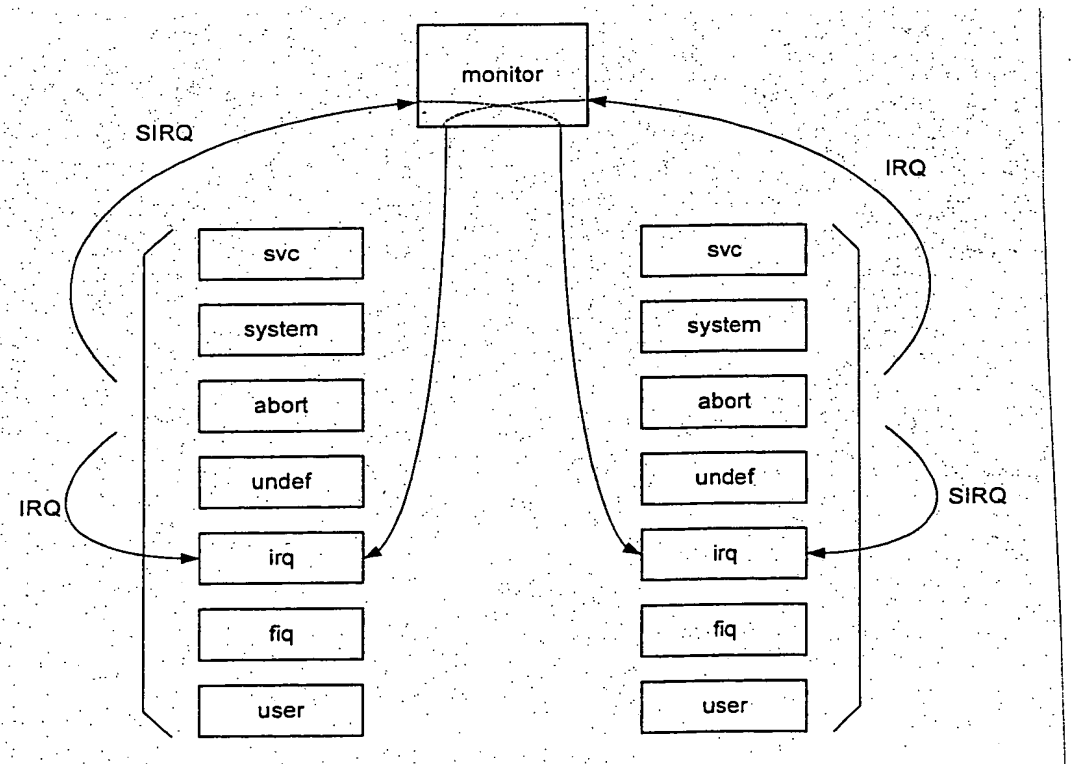


Fig. 10

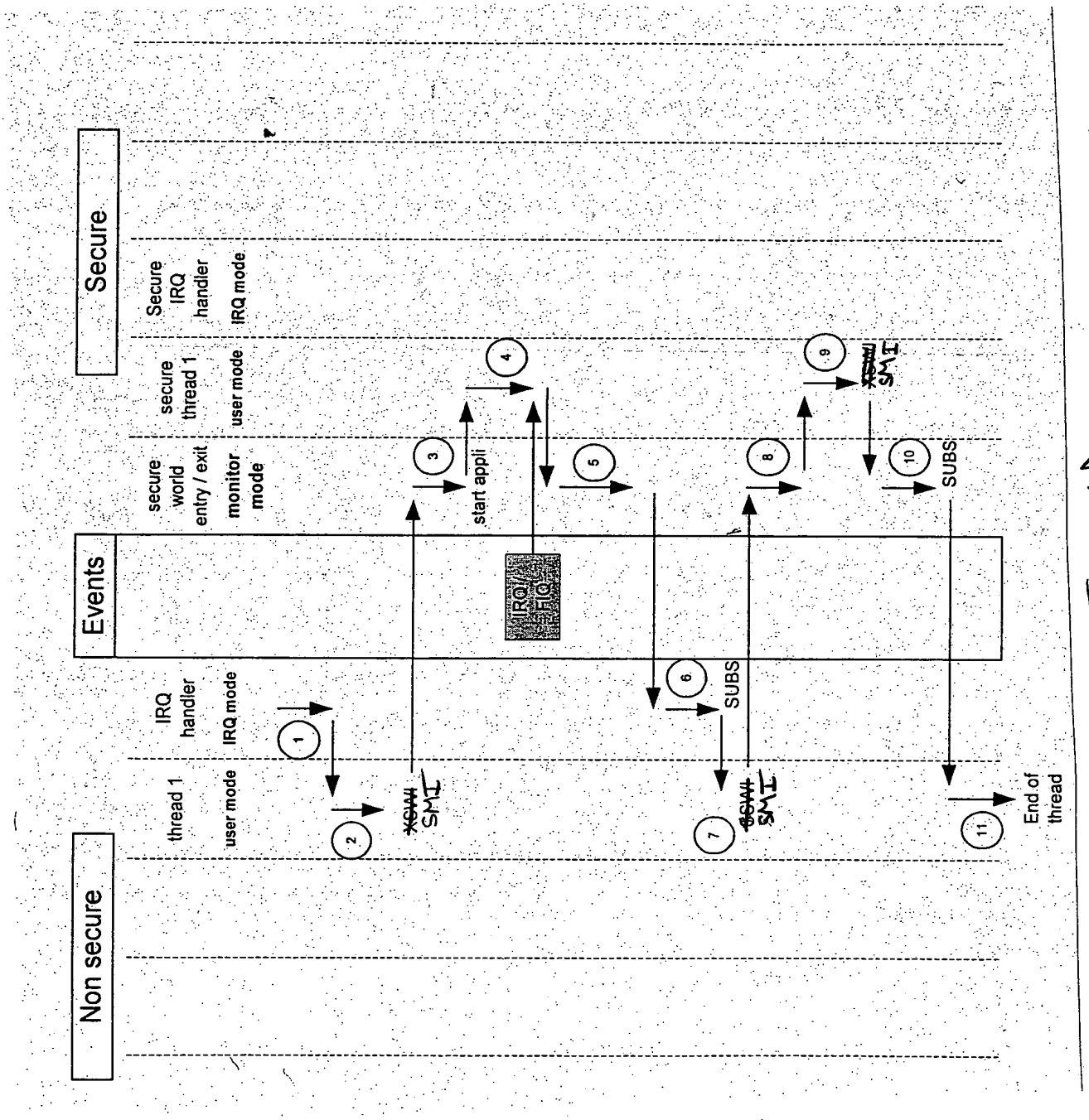


Fig 11A

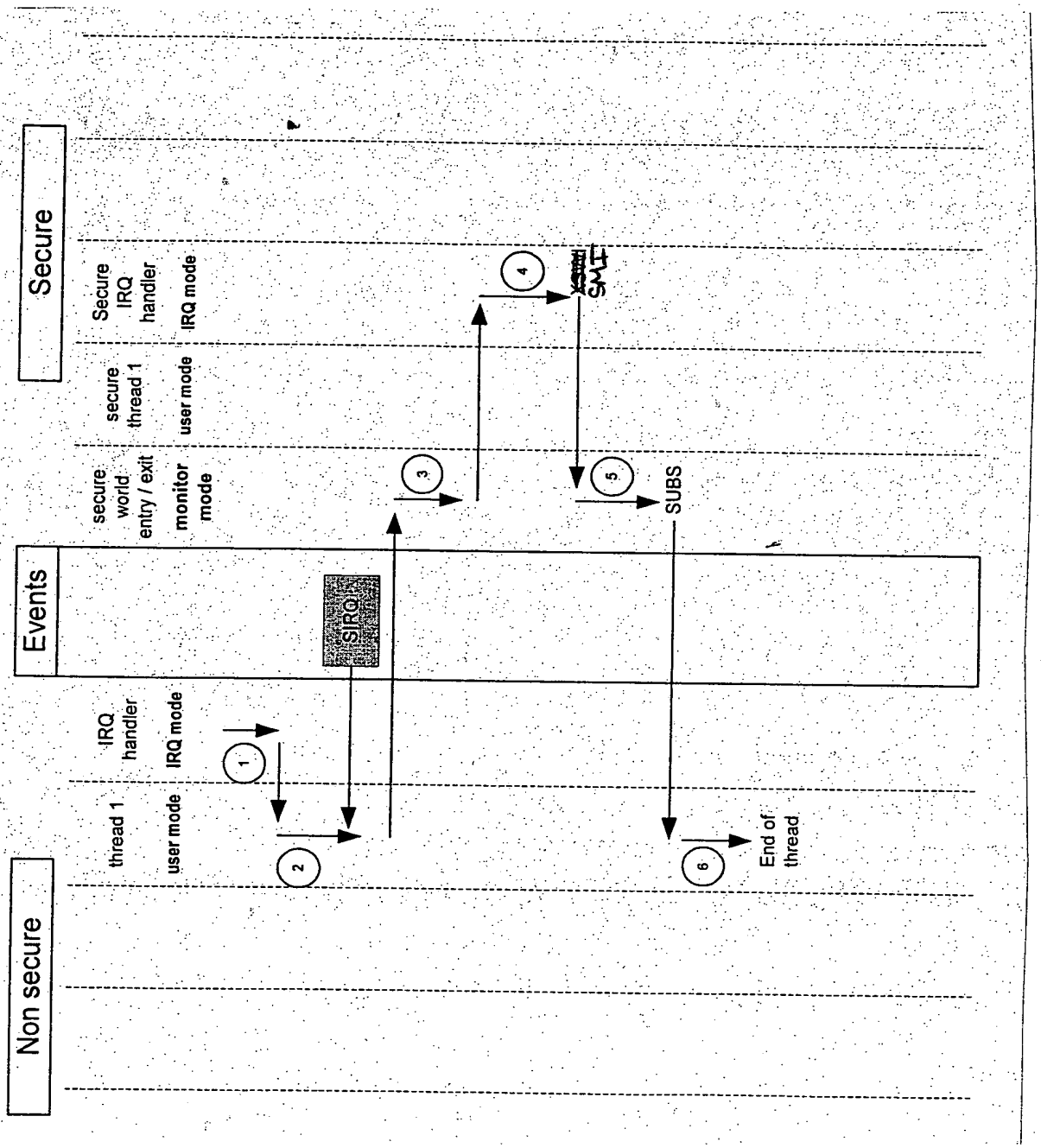


Fig 13

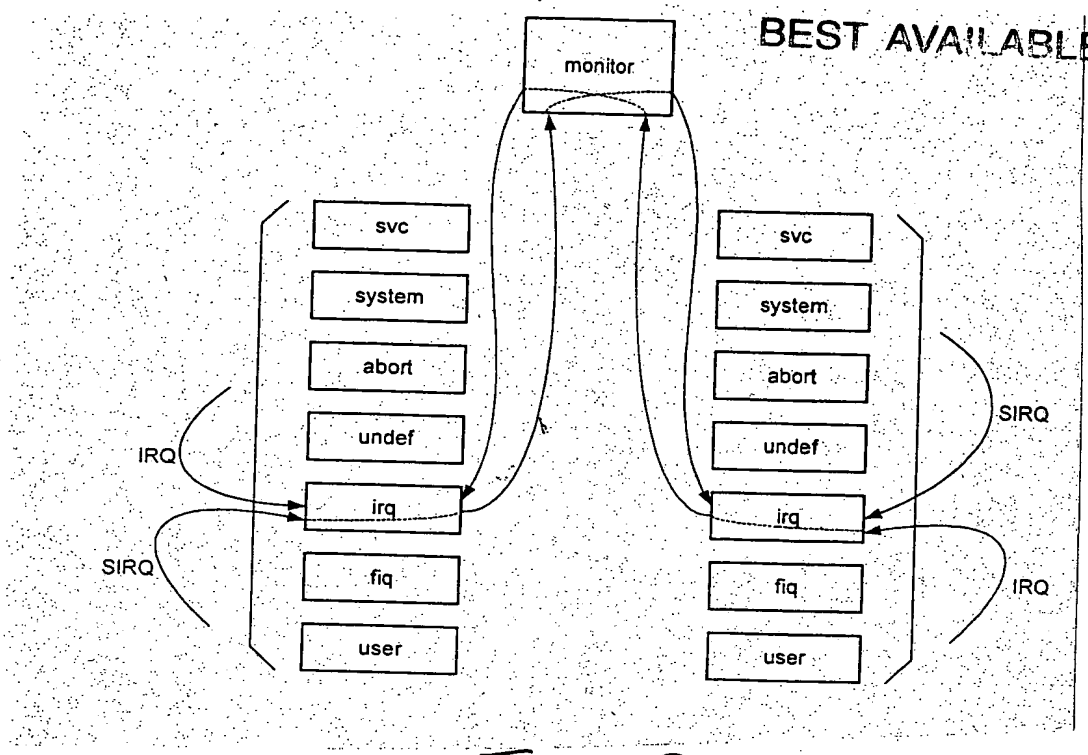


Fig. 12

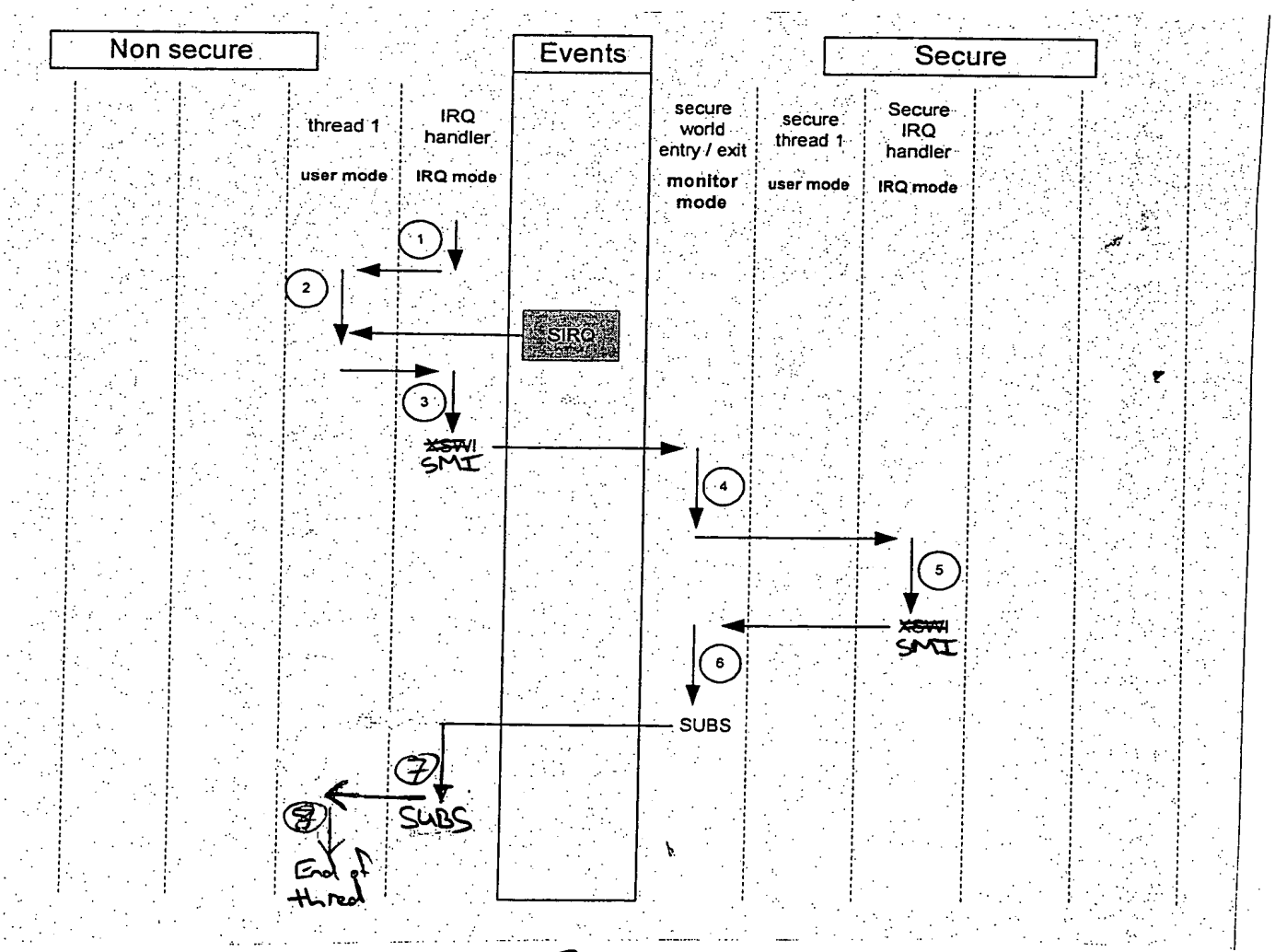


Fig. 13A

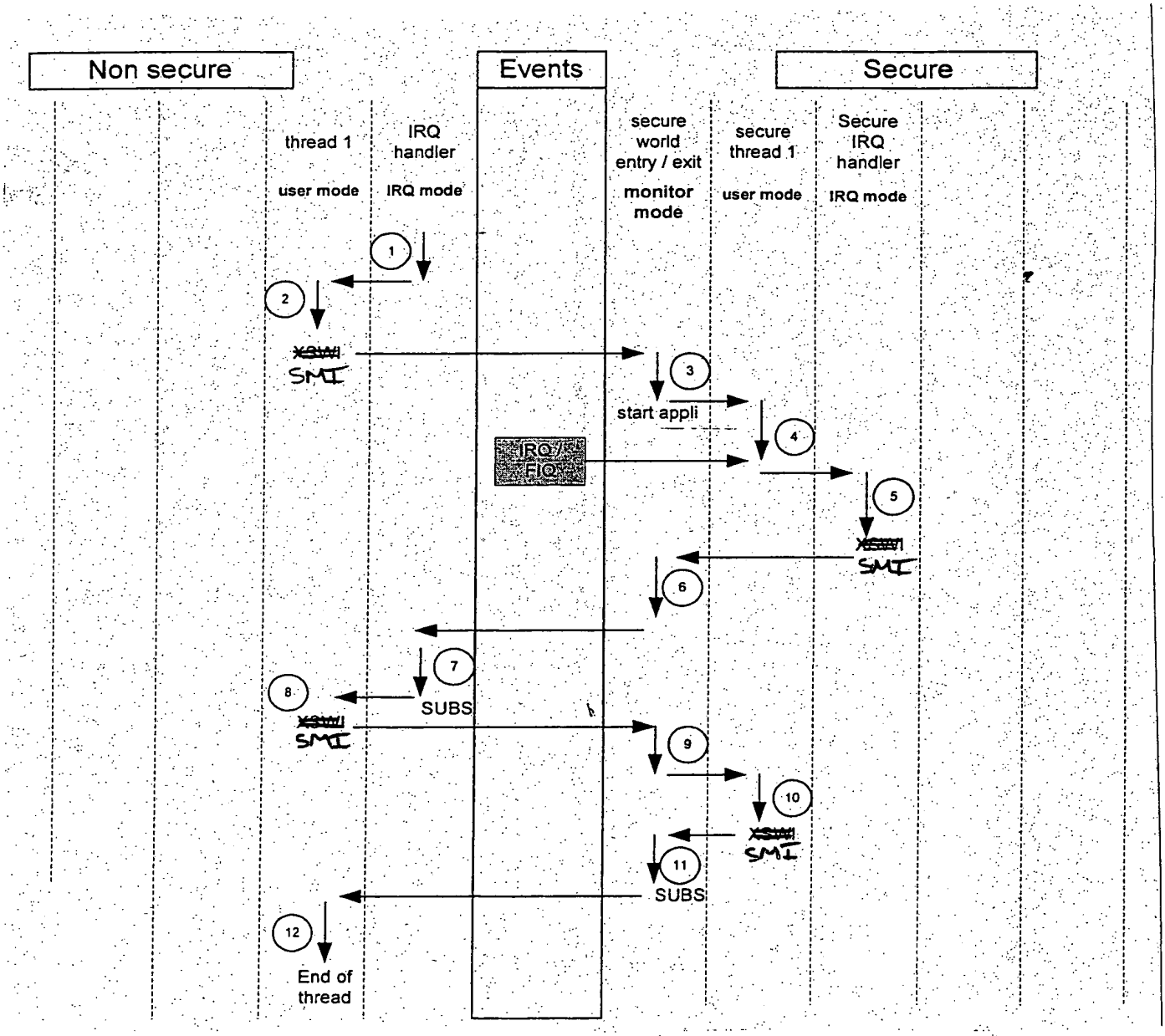


Fig. 13B

Exception	Vector offset	Corresponding mode
Reset	0x00	~~Supervisor~~ mode
Undef	0x04	Monitor mode / Undef mode
SWI	0x08	Supervisor mode / Monitor mode
Prefetch abort	0x0C	Abort mode / Monitor mode
Data abort	0x10	Abort mode / Monitor mode
IRQ / SIRQ	0x18	IRQ mode / Monitor mode
FIQ	0x1C	FIQ mode / Monitor mode
SMI	0x14	Undef mode / Monitor mode

Fig. 14

Monitor

Reset	VM0
Undef	VM1
SWI	VM2
Prefetch abort	VM3
Data abort	VM4
IRQ/SIRQ	VM5
FIQ	VM6
SMI	VM7

Secure

Reset	VS0
Undef	VS1
SWI	VS2
Prefetch abort	VS3
Data abort	VS4
IRQ/SIRQ	VS5
FIQ	VS6
SMI	VS7

Non-Secure

Reset	VNS0
Undef	VNS1
SWI	VNS2
Prefetch abort	VNS3
Data abort	VNS4
IRQ/SIRQ	VNS5
FIQ	VNS6
SMI	VNS7

Fig. 15

CP15 Exception Trap Mask Register

only NS world exceptions illustrated

0	1	1	1	1	0	1
Reset + SMI	SWI	Prefetch Abort	Data Abort	IRQ	SIRQ	FIQ

1 = Mon(S)

0 = NS

OR via hardware/external pin value

Fig 16

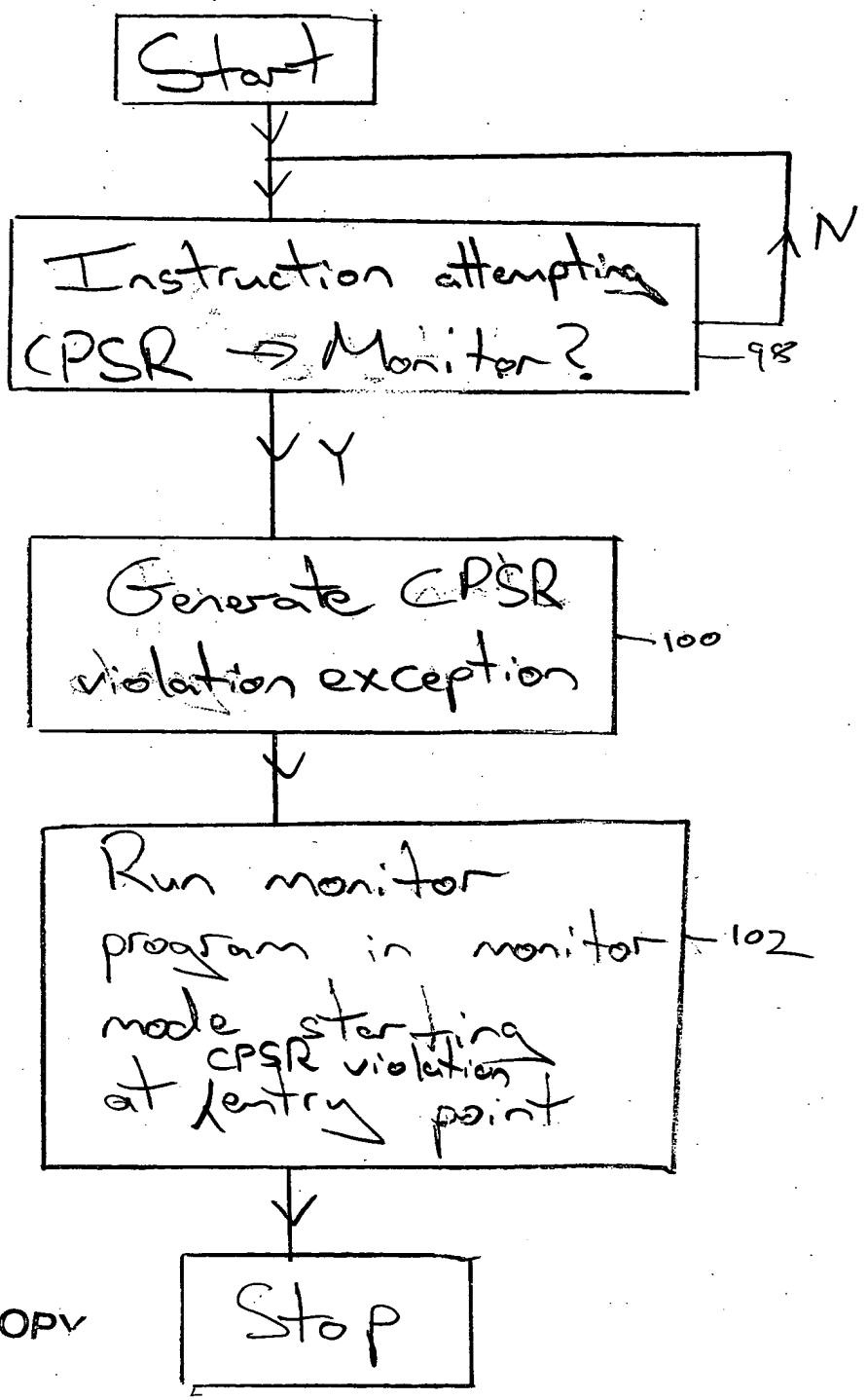


Fig. 17

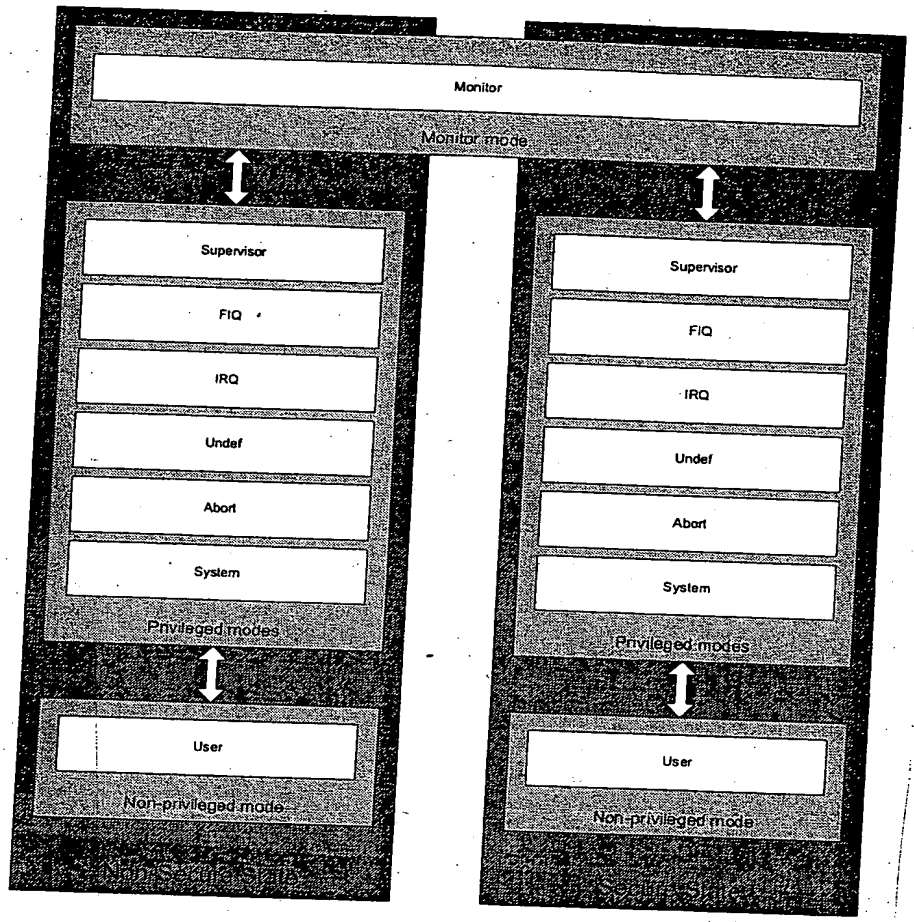


FIGURE 18

User	System	Supervisor	Abort	Undefined	Interrupt	Fast Interrupt
R0	R0	R0	R0	R0	R0	R0
R1	R1	R1	R1	R1	R1	R1
R2	R2	R2	R2	R2	R2	R2
R3	R3	R3	R3	R3	R3	R3
R4	R4	R4	R4	R4	R4	R4
R5	R5	R5	R5	R5	R5	R5
R6	R6	R6	R6	R6	R6	R6
R7	R7	R7	R7	R7	R7	R7
R8	R8	R8	R8	R8	R8	R8_fiq
R9	R9	R9	R9	R9	R9	R9_fiq
R10	R10	R10	R10	R10	R10	R10_fiq
R11	R11	R11	R11	R11	R11	R11_fiq
R12	R12	R12	R12	R12	R12	R12_fiq
R13	R13	R13_svc	R13_abt	R13_und	R13_irq	R13_fiq
R14	R14	R14_svc	R14_abt	R14_und	R14_irq	R14_fiq
PC	PC	PC	PC	PC	PC	PC

CPSR	CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
		SPSR_svc	SPSR_abt	SPSR_und	SPSR_irq	SPSR_fiq

Monitor
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13_mon
R14_mon
PC

CPSR
SPSR_mon

FIGURE 19

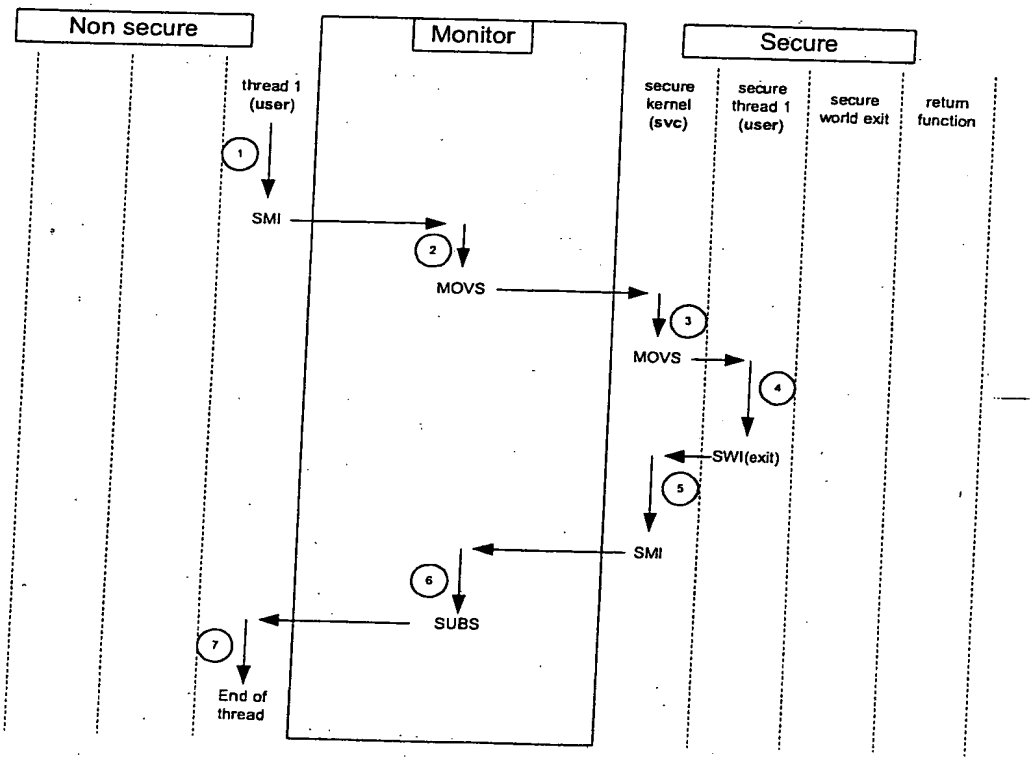


FIGURE 20

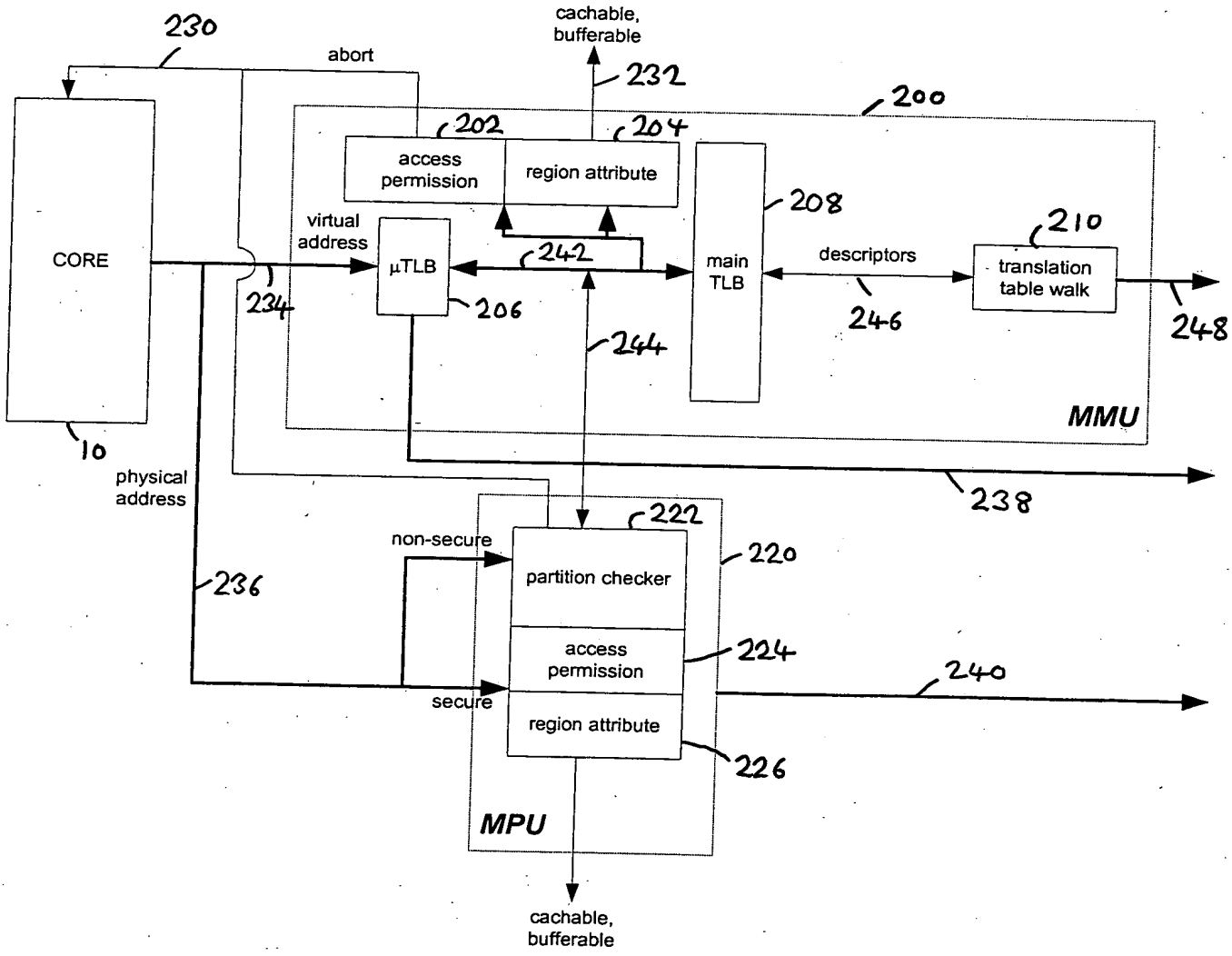


FIG. 21

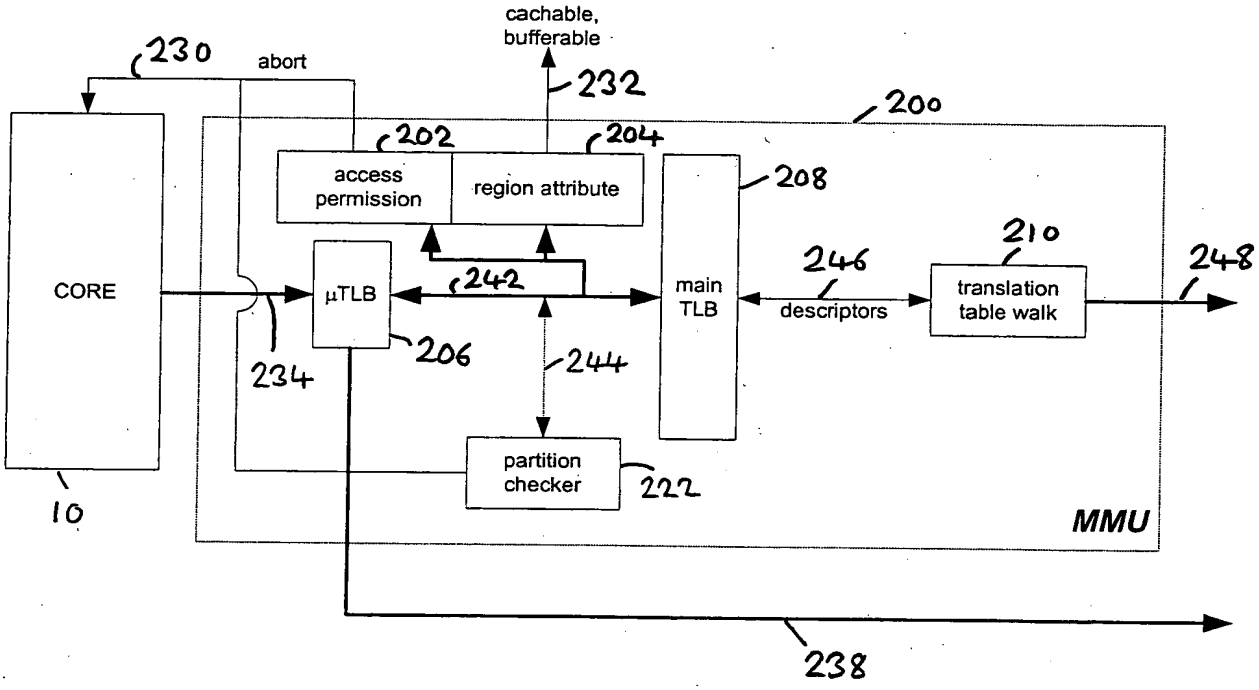


FIG. 22

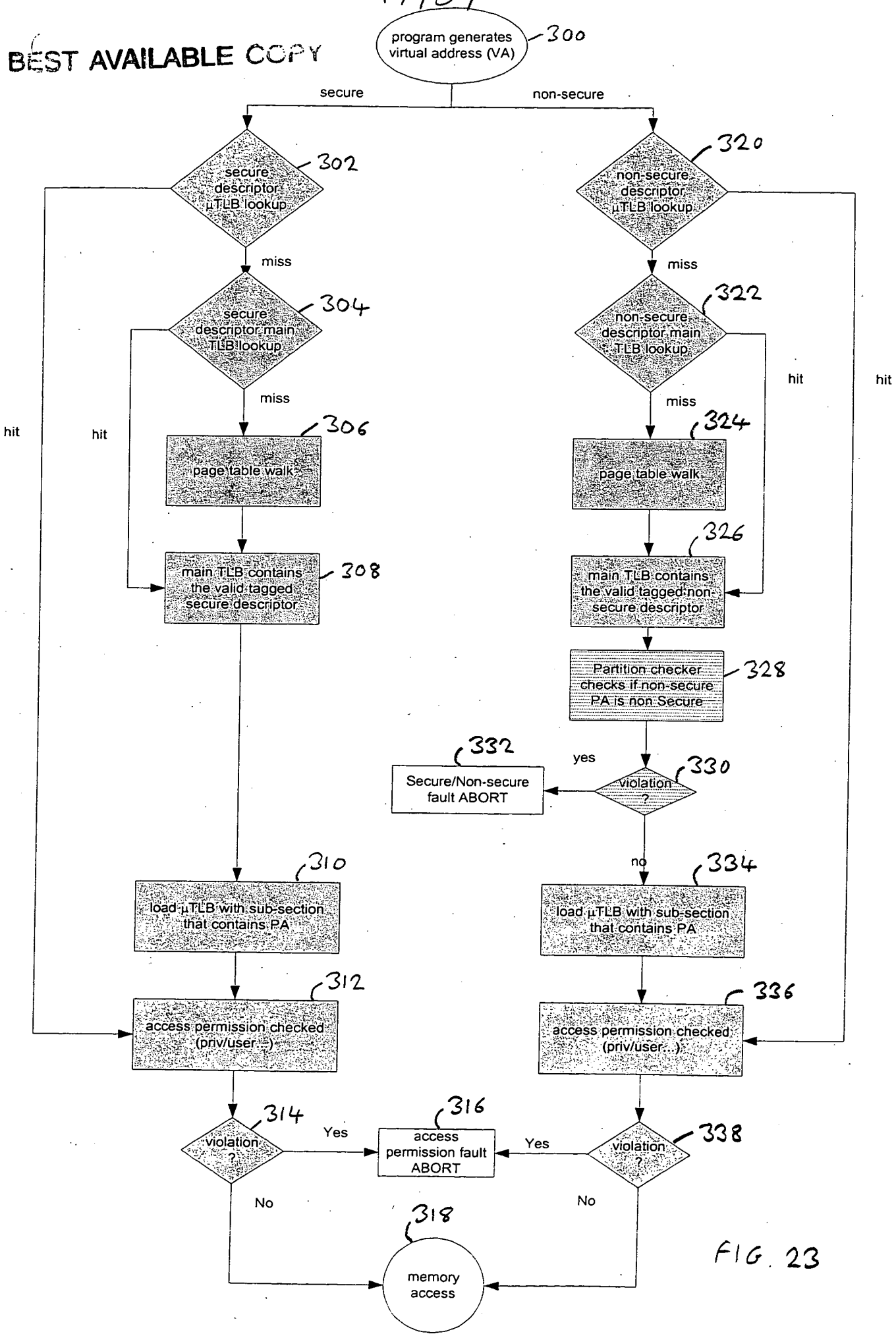


FIG. 23

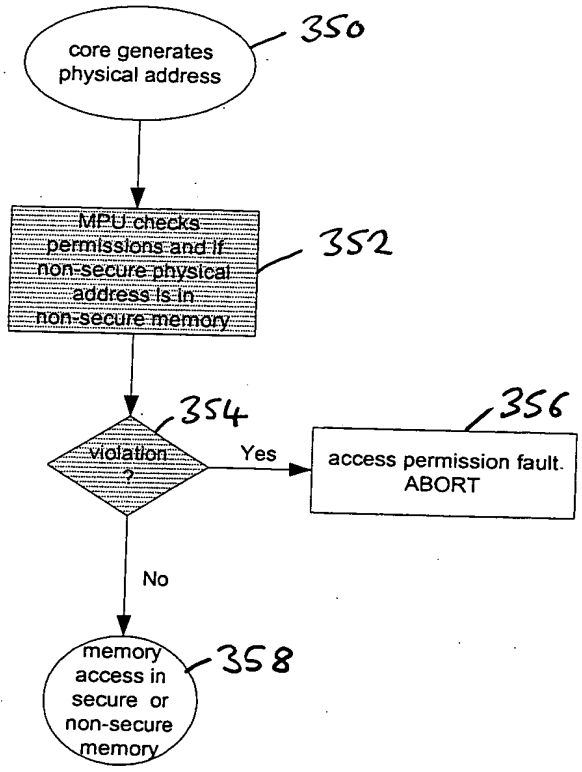

core generates physical address
350
MPU checks permissions and if non-secure physical address is in non-secure memory
352
354
violation ?
Yes
356
access permission fault. ABORT
No
memory access in secure or non-secure memory
358

FIG. 24

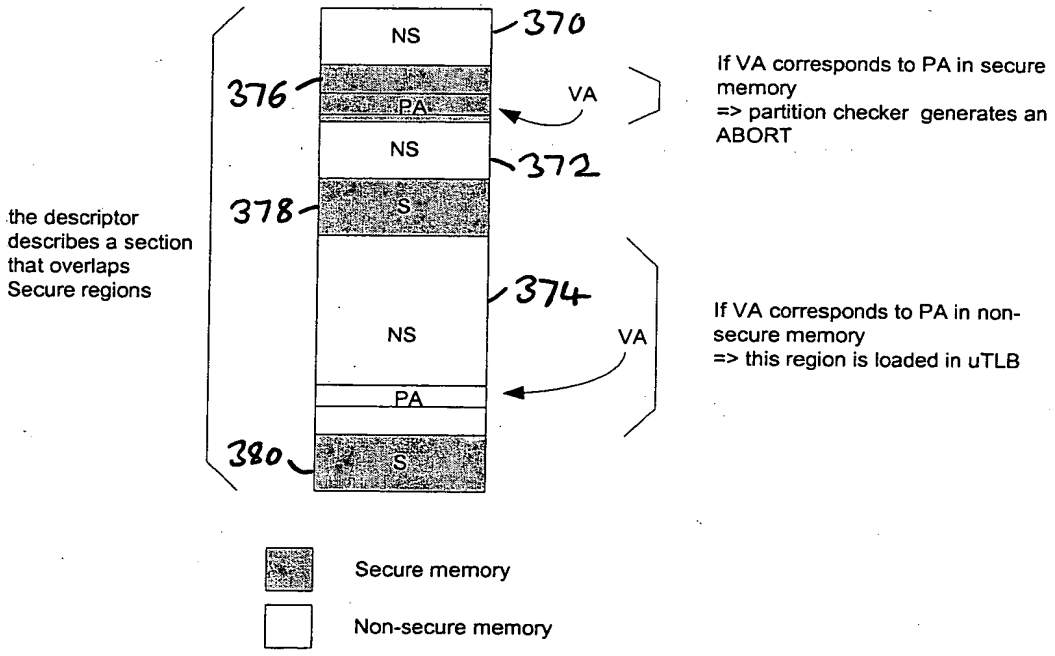

NS
370
376
PA
VA
If VA corresponds to PA in secure memory
=> partition checker generates an ABORT
NS
372
378
S
the descriptor describes a section that overlaps Secure regions
374
NS
VA
If VA corresponds to PA in non-secure memory
=> this region is loaded in uTLB
PA
380
S
Secure memory
Non-secure memory

FIG. 25

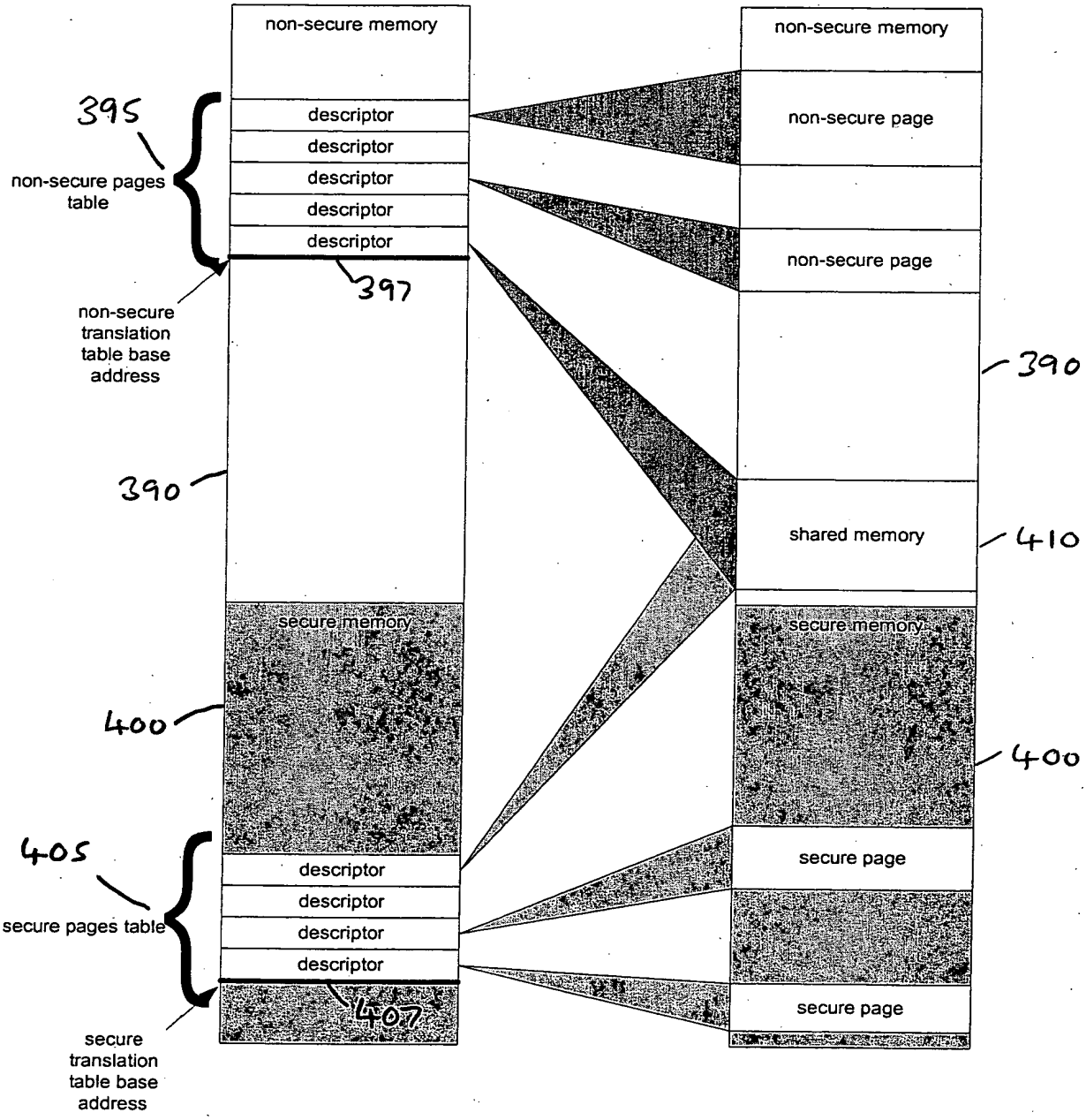

non-secure memory
395
descriptor
descriptor
descriptor
descriptor
descriptor
non-secure pages table
397
non-secure translation table base address
390
secure memory
400
405
descriptor
descriptor
descriptor
descriptor
secure pages table
407
secure translation table base address
non-secure memory
non-secure page
non-secure page
390
shared memory
410
secure memory
400
secure page
secure page

FIG. 26

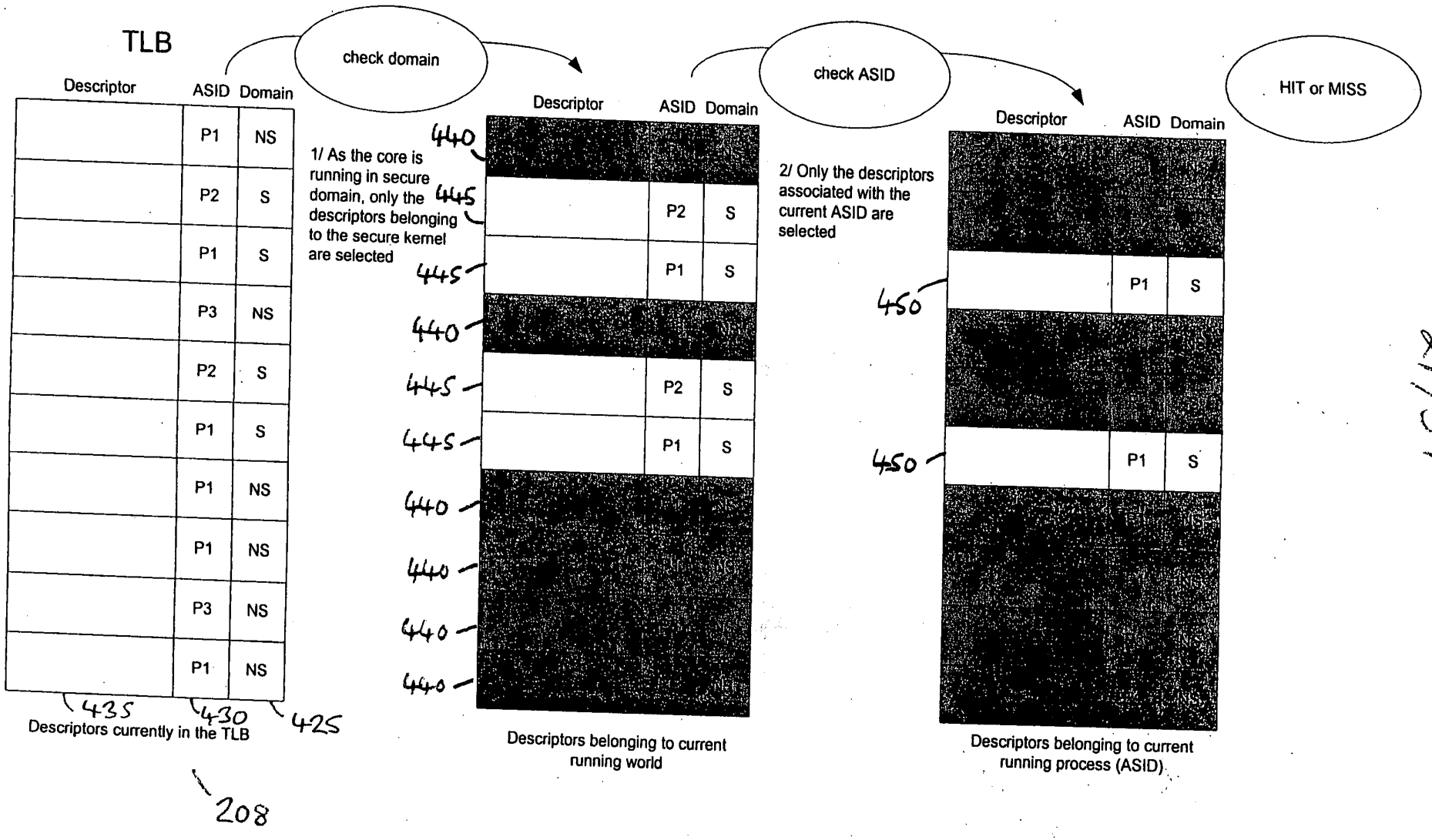

TLB
Descriptor
ASID
Domain
P1 NS
P2 S
P1 S
P3 NS
P2 S
P1 S
P1 NS
P1 NS
P3 NS
P1 NS
435
430
425
Descriptors currently in the TLB
208
check domain
1/ As the core is running in secure domain, only the descriptors belonging to the secure kernel are selected
440
445
Descriptor
ASID
Domain
P2 S
P1 S
Descriptors belonging to current running world
check ASID
2/ Only the descriptors associated with the current ASID are selected
450
HIT or MISS
Descriptor
ASID
Domain
P1 S
Descriptors belonging to current running process (ASID)

FIG. 27

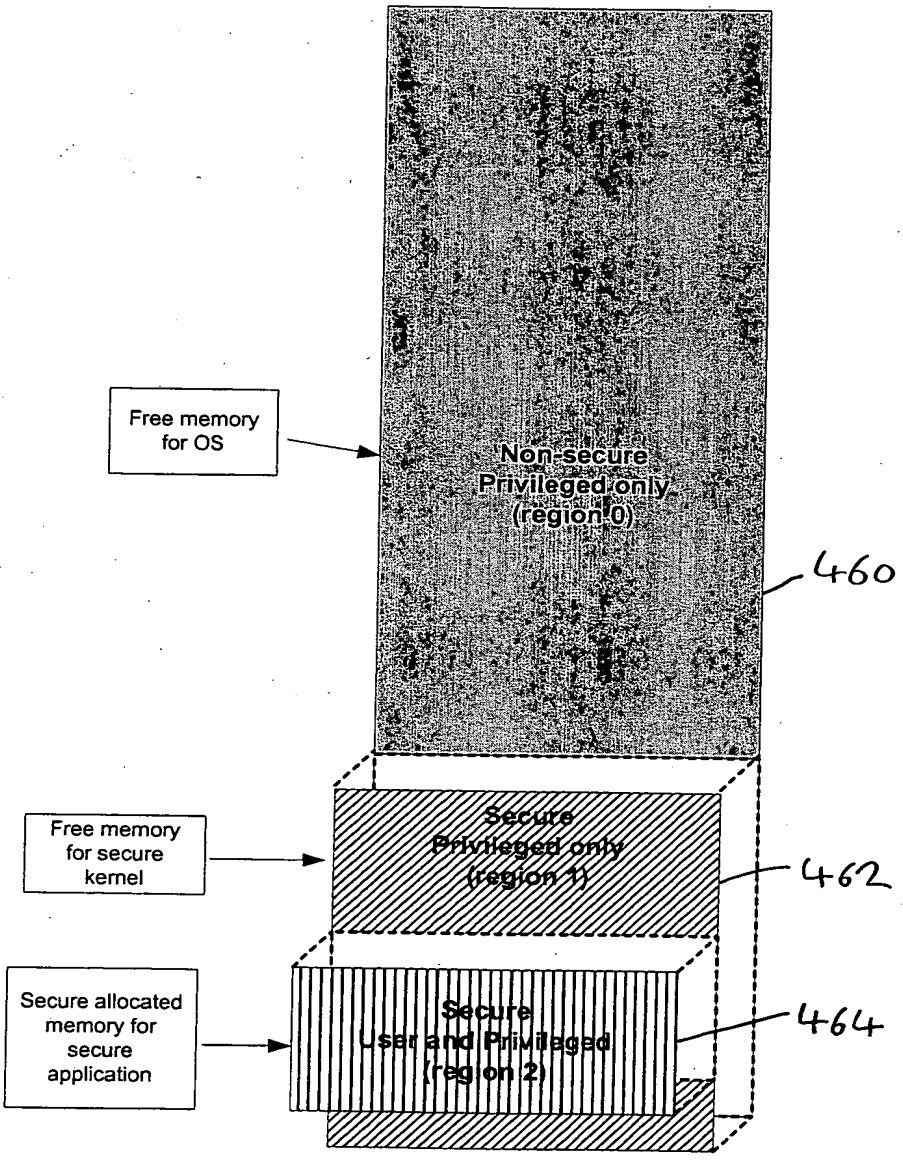

Free memory for OS
Non-secure Privileged only (region 0)
460
Free memory for secure kernel
Secure Privileged only (region 1)
462
Secure allocated memory for secure application
Secure User and Privileged (region 2)
464

FIG. 28

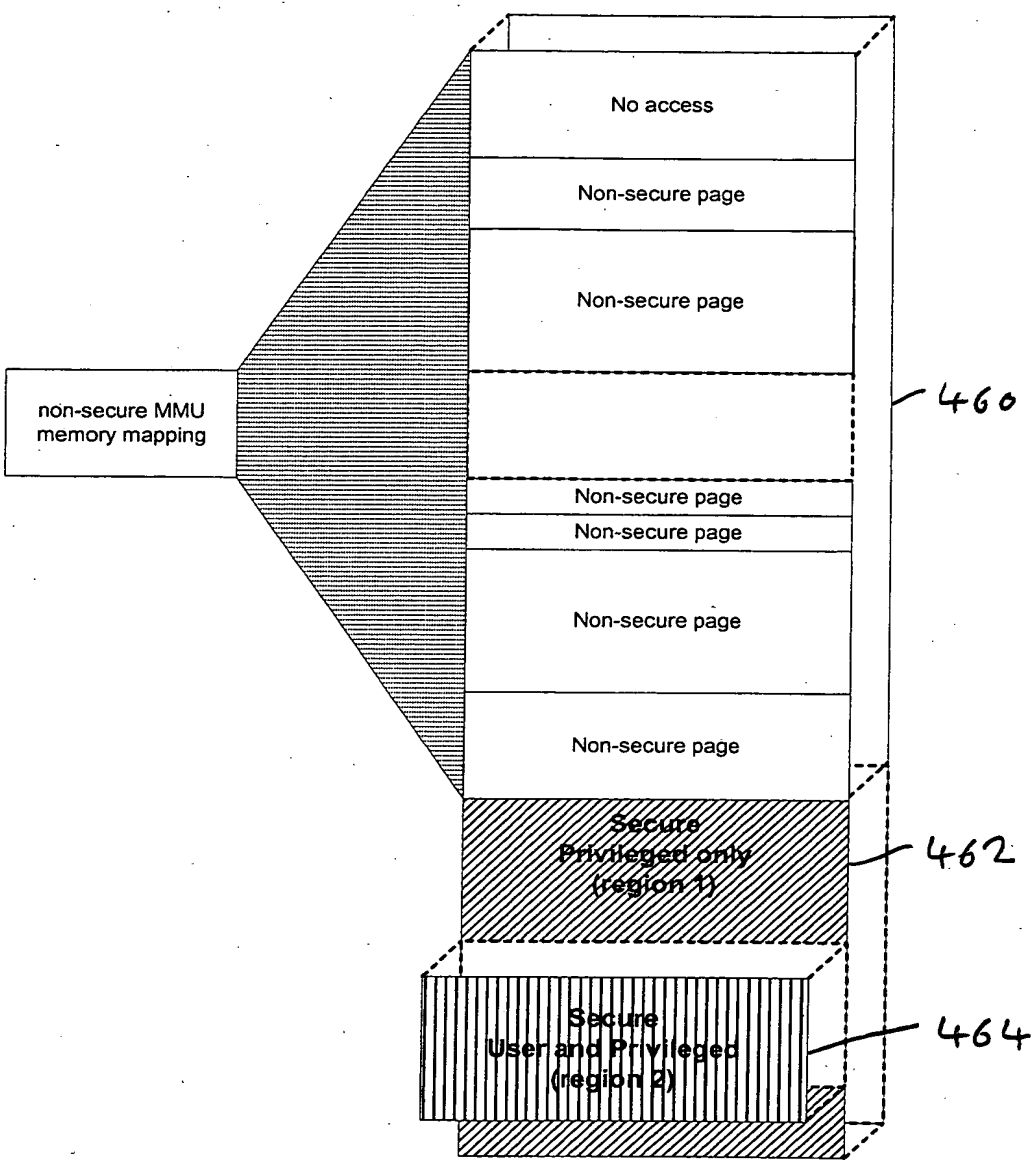

No access
Non-secure page
Non-secure page
non-secure MMU
memory mapping
460
Non-secure page
Non-secure page
Non-secure page
Non-secure page
Secure
Privileged only
(region 1)
462
Secure
User and Privileged
(region 2)
464

FIG. 29

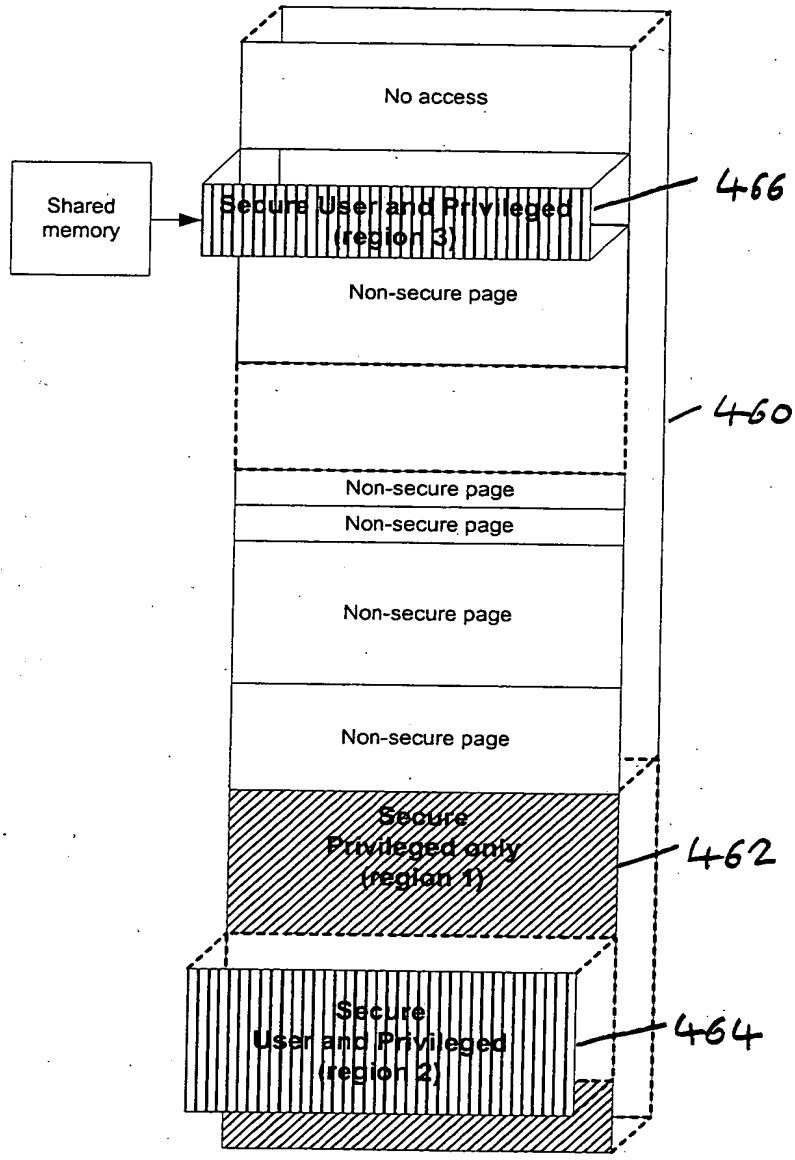

No access
Shared memory
Secure User and Privileged (region 3)
466
Non-secure page
460
Non-secure page
Non-secure page
Non-secure page
Non-secure page
Secure Privileged only (region 1)
462
Secure User and Privileged (region 2)
464

FIG. 30

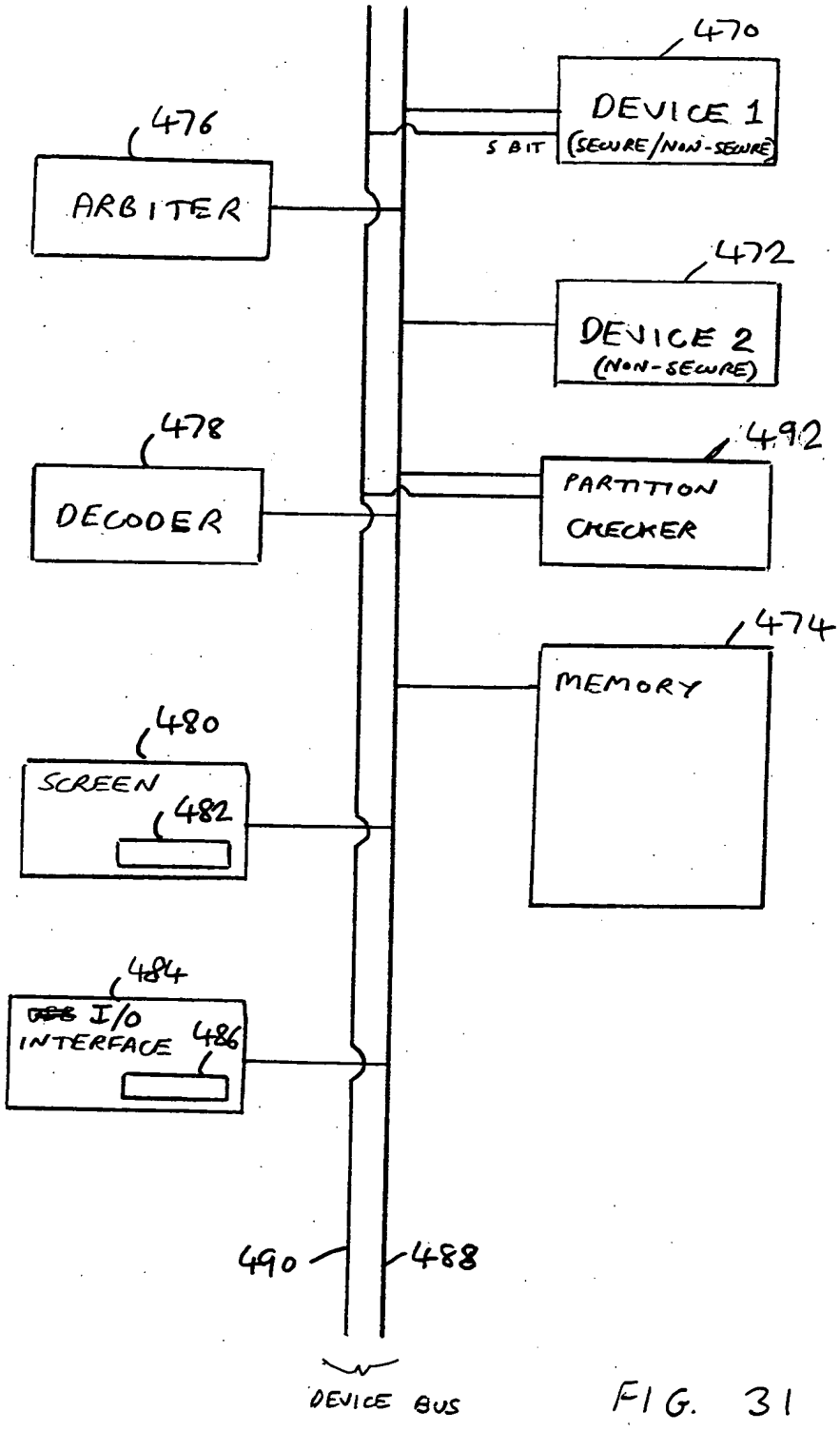

470
DEVICE 1
(SECURE/NON-SECURE)
5 BIT
476
ARBITER
472
DEVICE 2
(NON-SECURE)
478
DECODER
492
PARTITION
CHECKER
474
MEMORY
480
SCREEN
482
484
I/O
INTERFACE
486
490
488
DEVICE BUS

FIG. 31

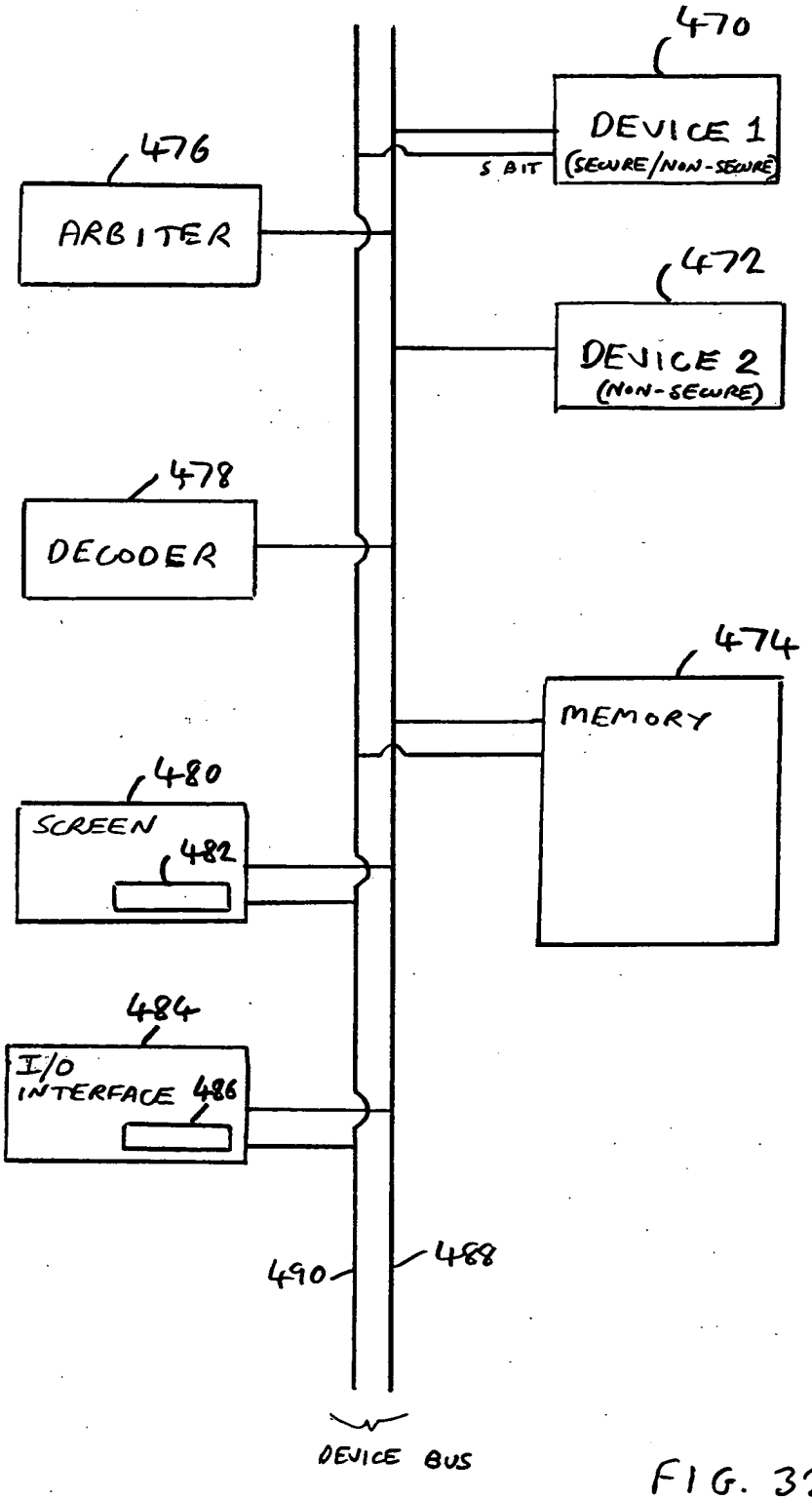

470
DEVICE 1
(SECURE/NON-SECURE)
5 BIT
476
ARBITER
472
DEVICE 2
(NON-SECURE)
478
DECODER
474
MEMORY
480
SCREEN
482
484
I/O
INTERFACE
486
490
488
DEVICE BUS

FIG. 32

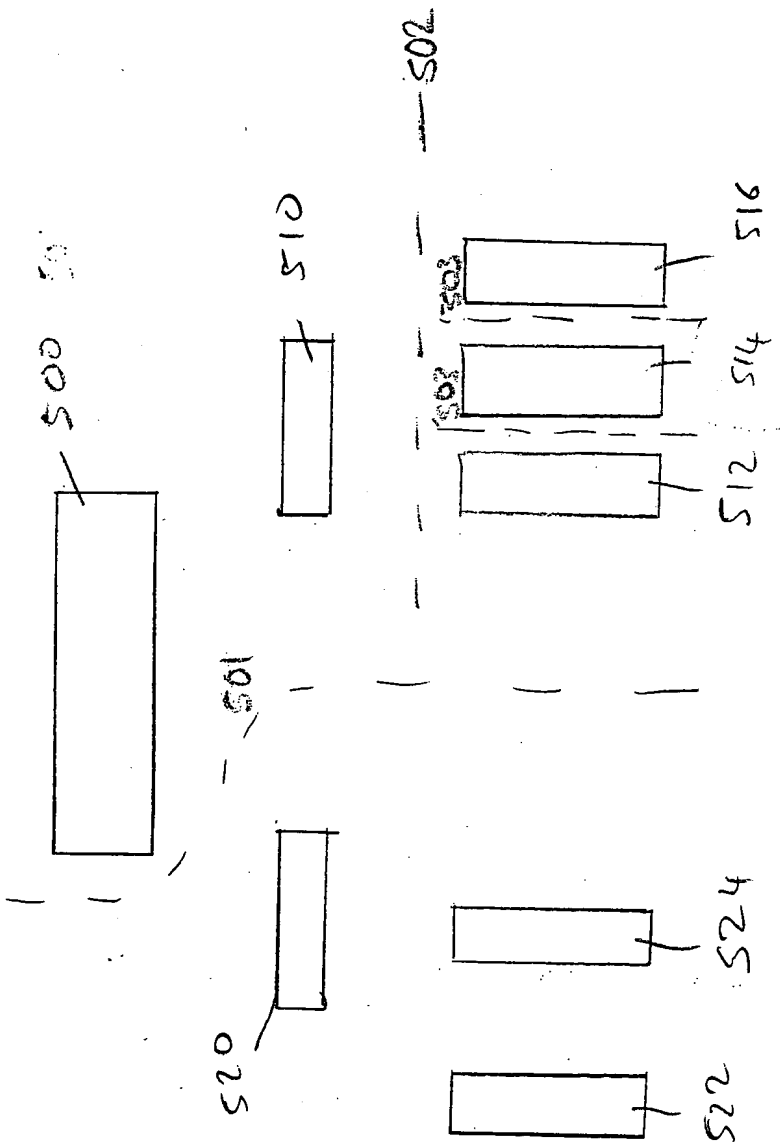


Figure 33

Method of entry	How to program?	How to enter?	Entry mode
Breakpoint hits	Debug TAP or software (CP14)	Program breakpoint register and/or context-ID register and comparisons succeed with Instruction Address and/or CP15 Context ID ([2]).	Halt/monitor ([1])
Software breakpoint instruction	Put a BKPT instruction into scan chain 4 (Instruction Transfer Register) through Debug TAP or Use BKPT instruction directly in the code.	BKPT instruction must reach execution stage.	Halt/monitor
Vector trap breakpoint	Debug TAP	Program vector trap register and address matches.	Halt/monitor
Watchpoint hits	Debug TAP or software (CP14)	Program watchpoint register and/or context-ID register and comparisons succeed with Instruction Address and/or CP15 Context ID ([2]).	Halt/monitor ([1])
Internal debug request	Debug TAP	Halt instruction has been scanned in.	Halt
External debug request	Not applicable	EDBGRQ input pin is asserted.	Halt

([1]): In monitor mode, breakpoints and watchpoints cannot be data-dependent.

([2]): The cores have support for thread-aware breakpoints and watchpoints in order to enable secure debug on some particular threads.

Figure 34

Name	Meaning	Reset value	Access	Inserted in scan chain for test
Monitor mode enable bit	0: halt mode 1: monitor mode	1	R/W by programming the ICE by the JTAG (scan1) ▪ R/W by using MRC/MCR instruction (CP14)	yes
Secure debug enable bit	0: debug in non-secure world only. 1: debug in secure world and non-secure world	0	In functional mode or debug monitor mode: R/W by using MRC/MCR instruction (CP14) (only in secure supervisor mode) In Debug halt mode: No access – MCR/MRC instructions have any effect. (R/W by programming the ICE by the JTAG (scan1) if JSDAEN=1	no
Secure trace enable bit	0: ETM is enabled in non-secure world only. 1: ETM is enabled in secure world and non-secure world	0	In functional mode or debug monitor mode: R/W by using MRC/MCR instruction (CP14) (only in secure supervisor mode) In Debug halt mode: No access – MCR/MRC instructions have any effect. (R/W by programming the ICE by the JTAG (scan1) if JSDAEN=1	no
Secure user-mode enable bit	0: debug is not possible in secure user mode 1: debug is possible in secure user mode	1	In functional mode or debug monitor mode: R/W by using MRC/MCR instruction (CP14) (only in secure supervisor mode) In Debug halt mode: No access – MCR/MRC instructions have any effect. (R/W by programming the ICE by the JTAG (scan1) if JSDAEN=1	no
Secure thread-aware enable bit	0: debug is not possible for a particular thread 1: debug is possible for a particular thread	0	In functional mode or debug monitor mode: R/W by using MRC/MCR instruction (CP14) (only in secure supervisor mode) In Debug halt mode: No access – MCR/MRC instructions have any effect. (R/W by programming the ICE by the JTAG (scan1) if JSDAEN=1	no

Figure 35

Function Table

D	CK	Q[n+1]
0	⎍	0
1	⎍	1
X	⎎	Q[n]

Logic Symbol

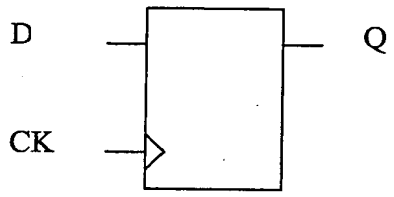


FIGURE 36

Function Table

D	SI	SE	CK	Q[n+1]
0	X	0	↑	0
1	X	0	↑	1
X	X	X	↓	Q[n]
X	0	1	↑	0
X	1	1	↑	1

Logic Symbol

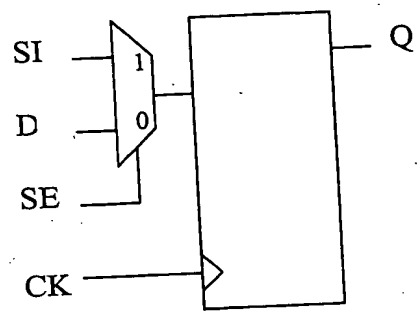


Figure 37

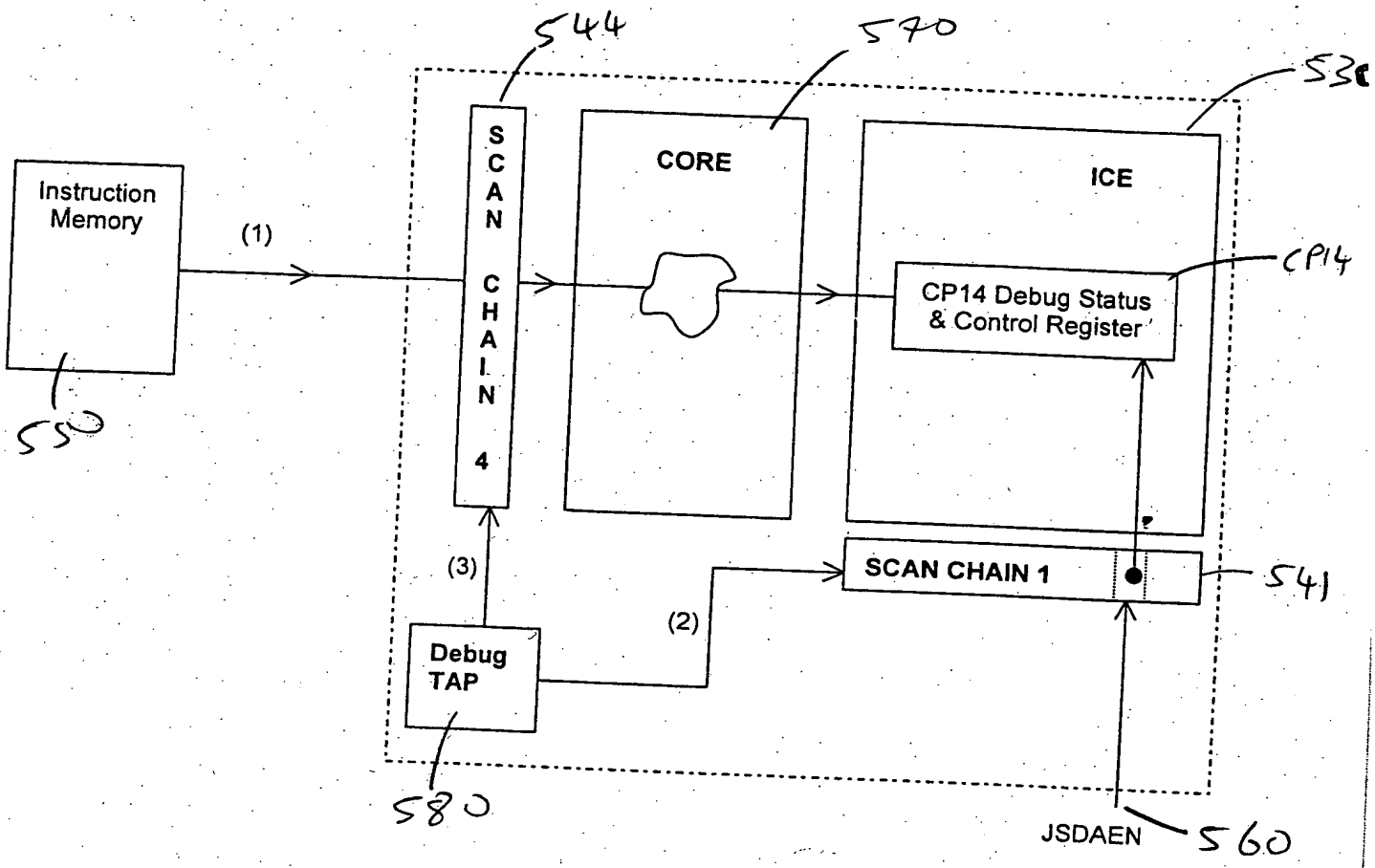

544
570
Instruction Memory
(1)
SCAN CHAIN 4
CORE
ICE
CP14 Debug Status & Control Register
CP14
550
(3)
(2)
SCAN CHAIN 1
541
Debug TAP
580
JSDAEN
560

Figure 41

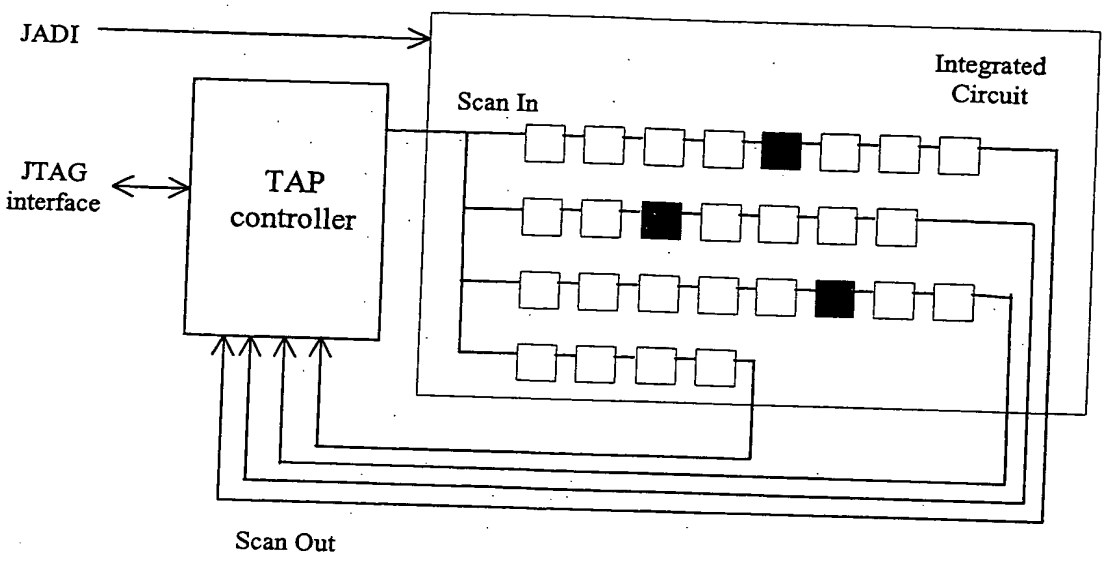


FIGURE 40A

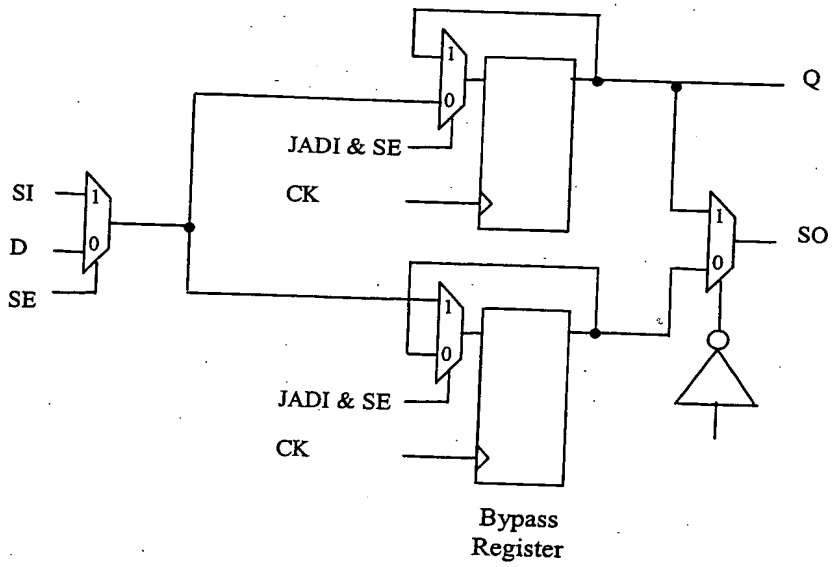


FIGURE 40B

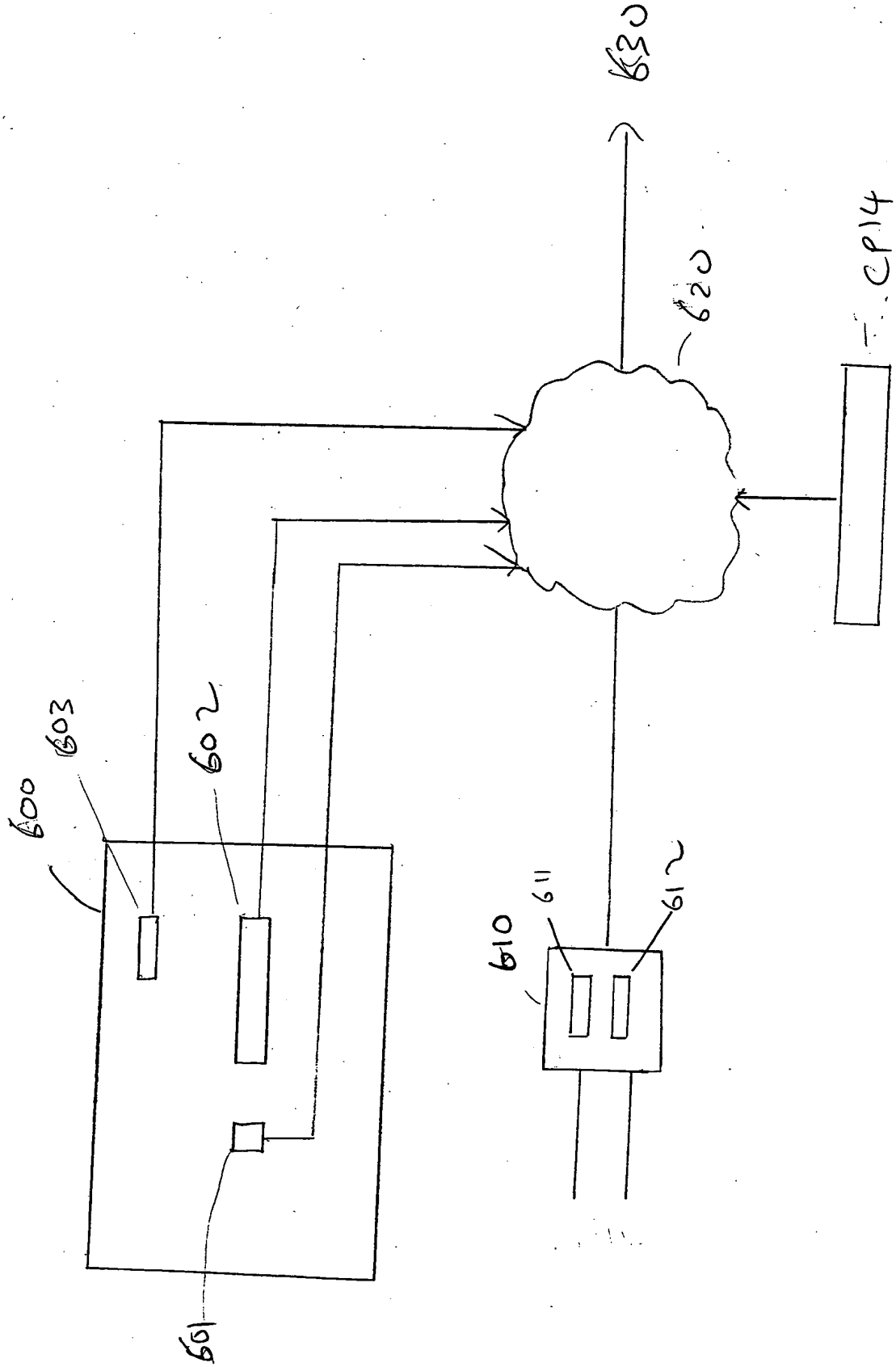


Figure 42

CP14 bits in Debug and Status Control register			meaning
Secure debug enable bit	Secure user-mode debug enable bit	Secure thread-aware debug enable bit	
0	X	X	No intrusive debug in entire secure world is possible. Any debug request, breakpoints, watchpoints, and other mechanism to enter debug state are ignored in entire secure world.
1	0	X	Debug in entire secure world is possible
1	1	0	Debug in secure user-mode only. Any debug request, breakpoints, watchpoints, and other mechanism to enter debug state are taken into account in user mode only. (Breakpoints and watchpoints linked or not to a thread ID are taken into account). Access in debug is restricted to what secure user can have access to.
1	1	1	Debug is possible only in some particular threads. In that case only thread-aware breakpoints and watchpoints linked to a thread ID are taken into account to enter debug state. Each thread can moreover debug its own code, and only its own code.

Figure 43A

CP14 bits in Debug and Status Control register			meaning
Secure trace enable bit	Secure user-mode debug enable bit	Secure thread-aware debug enable bit	
0	X	X	No observable debug in entire secure world is possible. Trace module (ETM) must not trace internal core activity.
1	0	X	Trace in entire secure world is possible
1	1	0	Trace is possible when the core is in secure user-mode only.
1	1	1	Trace is possible only when the core is executing some particular threads in secure user mode. Particular hardware must be dedicated for this, or re-use breakpoint register pair: Context ID match must enable trace instead of entering debug state.

Figure 43B

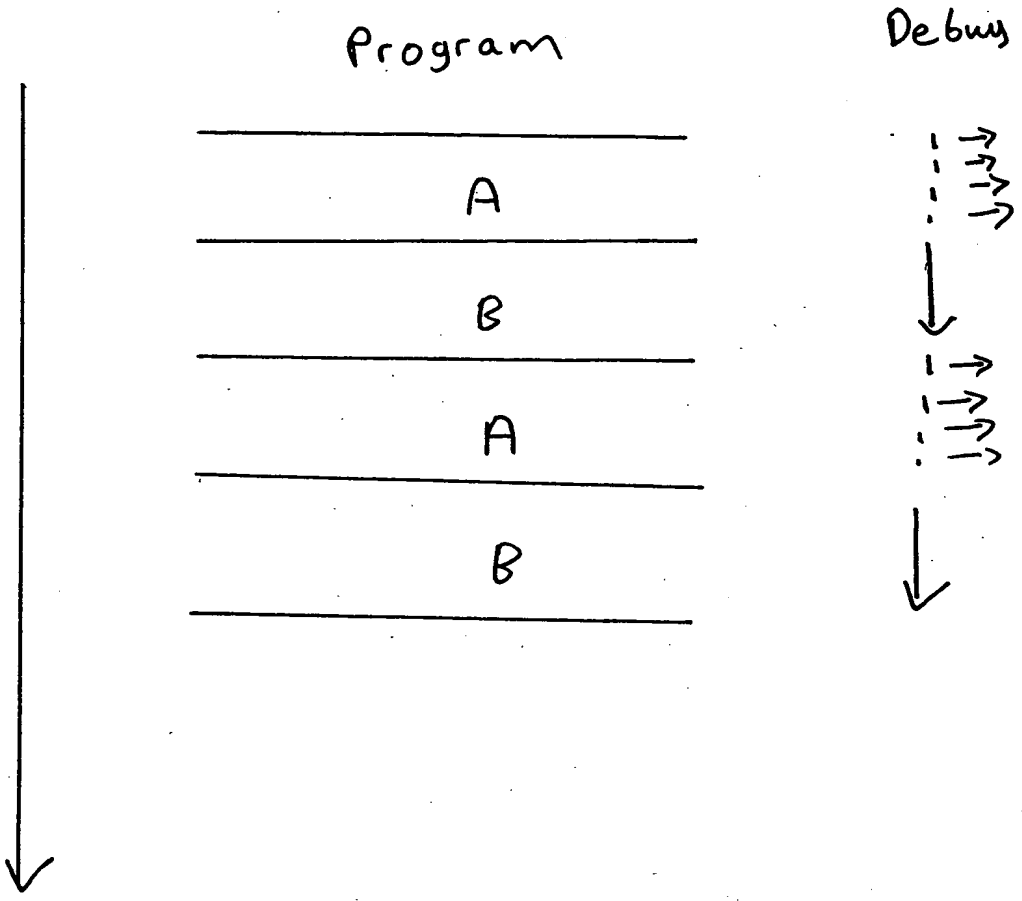


Figure 44

Method of entry	Entry when in non-secure world	entry when in secure world
Breakpoint hits	Non-secure prefetch abort handler	secure prefetch abort handler
Software breakpoint instruction	Non-secure prefetch abort handler	secure prefetch abort handler
Vector trap breakpoint	Disabled for non-secure data abort and non-secure prefetch abort interruptions. For other non-secure exceptions, prefetch abort.	Disabled for secure data abort and secure prefetch abort exceptions (1). For other exceptions, secure prefetch abort.
Watchpoint hits	Non-secure data abort handler	secure data abort handler
Internal debug request	Debug state in halt mode	debug state in halt mode
External debug request	Debug state in halt mode	debug state in halt mode

(1) See information on vector trap register,

(2) Note that when external or internal debug request is asserted, the core enters halt mode and not monitor mode.

Figure 45A

Method of entry	Entry in non-secure world	entry in secure world
Breakpoint hits	Non-secure prefetch abort handler	breakpoint ignored
Software breakpoint instruction	Non-secure prefetch abort handler	instruction ignored (1)
Vector trap breakpoint	Disabled for non-secure data abort and non-secure prefetch abort interruptions. For others interruption non-secure prefetch abort.	breakpoint ignored
Watchpoint hits	Non-secure data abort handler	watchpoint ignored
Internal debug request	Debug state in halt mode	request ignored
External debug request	Debug state in halt mode	request ignored
Debug re-entry from system speed access	not applicable	not applicable

(1) As substitution of BKPT instruction in secure world from non-secure world is not possible, non-secure abort must handle the violation.

Figure 45B